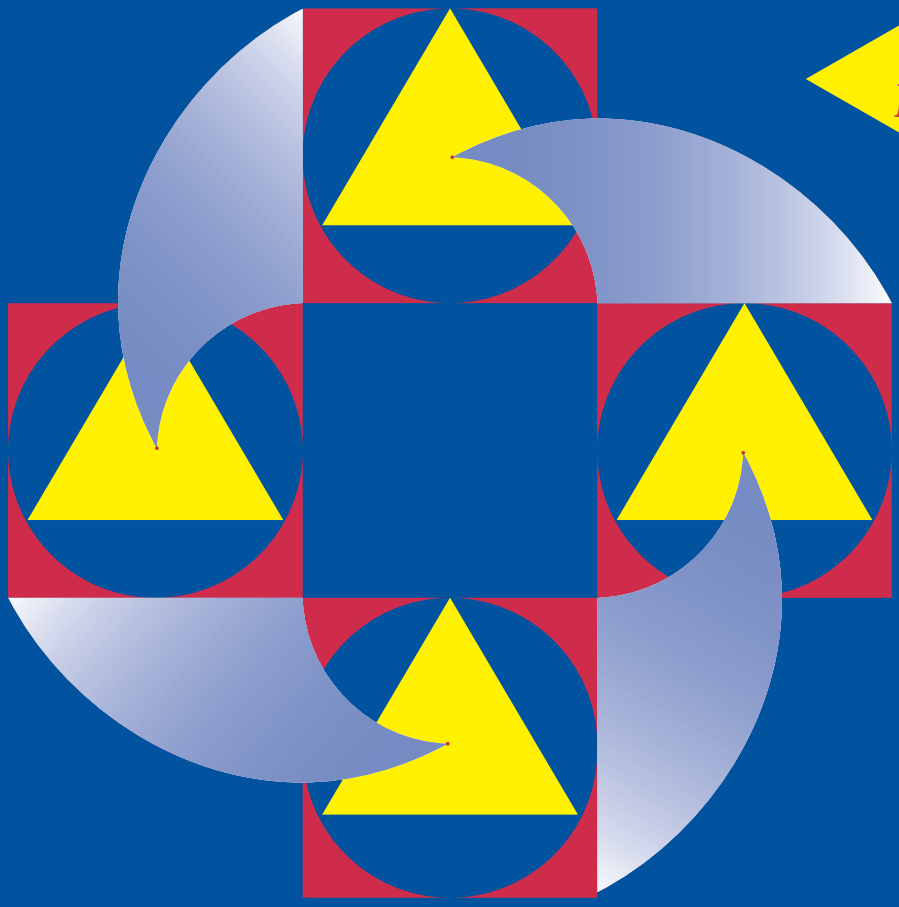
ALGORITMOS COMPUTACIONALES

Introducción al análisis y diseño

Tercera Edición



Baase
Van Gelder

Addison Wesley

Algoritmos computacionales

Introducción al análisis y diseño

TERCERA EDICIÓN

Sara Baase
San Diego State University

Allen Van Gelder
University of California at Santa Cruz

TRADUCCIÓN:

Roberto L. Escalona García
Universidad Nacional Autónoma de México

REVISIÓN TÉCNICA:

Saúl de la O. Torres
Escuela Superior de Cómputo
Instituto Politécnico Nacional

Pearson Educación ®

MÉXICO • ARGENTINA • BRASIL • COLOMBIA • COSTA RICA • CHILE
ESPAÑA • GUATEMALA • PERÚ • PUERTO RICO • VENEZUELA

Datos de catalogación bibliográfica

BAASE, SARA y GELDER ALLEN VAN
Algoritmos computacionales.
Introducción al análisis y diseño

PEARSON EDUCACIÓN, México, 2002

ISBN: 970-26-0142-8
Área: Universitarios

Formato: 18.5 × 23.5 cm Páginas: 704

ISBN 0-201-61244-5

Edición en español:
Editor: Guillermo Trujano Mendoza
e-mail: guillermo.trujano@pearsoned.com
Editor de desarrollo: Felipe de Jesús Castro Pérez
Supervisor de producción: José D. Hernández Garduño

Edición en inglés:
Acquisitions Editor: Maite Suarez-Rivas
Assistant Editor: Jason Miranda
Composition/Art: Paul C. Anagnostopoulos, Windfall Software
Copy Editor: Joan Flaherty
Proofreader: Brooke Albright
Cover Illustration: Janetmarie Colby
Cover Design: Lynne Reed
Manufacturing Coordinator: Timothy McDonald

TERCERA EDICIÓN, 2002

Calle 4 Núm. 25-2do. piso
Fracc. Industrial Alce Blanco
53370 Naucalpan de Juárez, Edo. de México
e-mail: editorial.universidades@pearsoned.com

Cámara Nacional de la Industria Editorial Mexicana Reg. Núm. 1031.

ISBN 970-26-0142-8

Impreso en México. *Printed in Mexico*.

1 2 3 4 5 6 7 8 9 0 - 04 03 02

Para Keith —siempre partícipe de lo que hago S.B.

Para mis padres, quienes fueron mis primeros maestros, y los más importantes A.V.G.

Prefacio

Objetivo

Este libro fue escrito para un curso completo sobre algoritmos; cuenta con suficiente material como para adoptar diversas orientaciones.

El objetivo del mismo incluye tres aspectos. Pretende enseñar algoritmos que se aplicarán en la resolución de problemas reales que se presentan a menudo en aplicaciones para computadora, enseñar principios y técnicas básicos de complejidad computacional (comportamiento de peor caso y caso promedio, consumo de espacio y cotas inferiores de la complejidad de un problema), e introducir las áreas de los problemas $\mathcal{NP}$-completos y los algoritmos paralelos.

Otra de las metas del libro, no menos importante que enseñar los temas que contiene, es desarrollar en el lector el hábito de siempre responder a un algoritmo nuevo con las preguntas: ¿Qué tan bueno es? ¿Hay una manera mejor? Por ello, en lugar de presentar una serie de algoritmos completos, "sacados de la manga", con su análisis, el libro normalmente comenta primero un problema, considera una o más estrategias para resolverlo (como podría hacer el lector que enfrenta el problema por primera vez) y luego comienza a desarrollar un algoritmo, lo analiza y lo modifica o lo rechaza hasta obtener un resultado satisfactorio. (Los enfoques alternativos que finalmente se rechazan también se examinan en los ejercicios; para el lector es útil saber por qué se les rechazó.)

Preguntas del tipo de ¿Cómo puede hacerse esto de forma más eficiente? ¿Qué estructura de datos sería útil en este caso? ¿En qué operaciones debemos concentrarnos para analizar este algoritmo? ¿Qué valor inicial debe asignarse a esta variable (o estructura de datos)?, aparecen a menudo en todo el texto. Por lo general damos la respuesta inmediatamente después de la pregunta, pero sugerimos a los lectores hacer una pausa antes de continuar la lectura y tratar de idear su propia respuesta. El aprendizaje no es un proceso pasivo.

Tenemos la esperanza de que los lectores también aprendan a visualizar cómo se comporta en la realidad un algoritmo con diversas entradas; es decir, ¿Qué ramas sigue? ¿Qué patrón de crecimiento y encogimiento siguen las pilas? ¿Cómo afecta al comportamiento presentar las entradas en diferentes formas (por ejemplo, enumerando los vértices o aristas de un grafo en distintos órdenes)? Tales preguntas se plantean en algunos de los ejercicios, pero no hacemos hincapié en ellas en el texto porque requieren un estudio minucioso de los pormenores de un gran número de ejemplos.

Casi todos los algoritmos que presentamos tienen utilidad práctica; decidimos no hacer hincapié en los que tienen un buen comportamiento asintótico pero no se desempeñan bien con entradas de tamaño útil (aunque sí incluimos algunos por su importancia). Los algoritmos específicos se escogieron por diversas razones que incluyen la importancia del problema, la ilustración de téc-

nicas de análisis, la ilustración de técnicas (como la búsqueda primero en profundidad) que dan pie a numerosos algoritmos, y la ilustración del desarrollo y mejoramiento de técnicas y algoritmos (como los programas Unión-Hallar).

Requisitos previos

El libro supone que el lector está familiarizado con estructuras de datos como listas ligadas, pilas y árboles, también asume que ha tenido contacto con la recursión. No obstante, incluimos un repaso, con especificaciones, de las estructuras de datos estándar y de algunas especializadas. También añadimos un repaso de la recursión que los estudiantes no deberán tener problemas para entender.

En el análisis de algoritmos utilizamos propiedades sencillas de los logaritmos y algo de cálculo (diferenciación para determinar el orden asintótico de una función e integración para aproximar sumatorias), aunque prácticamente no se usa cálculo más allá del capítulo 4. Hemos visto que muchos estudiantes se asustan al ver el primer logaritmo o signo de integral porque ha pasado un año o más desde su último curso de cálculo. Los lectores sólo necesitarán unas cuantas propiedades de los logaritmos y unas cuantas integrales del primer semestre de cálculo. En la sección 1.3 se repasan algunos de los temas necesarios de matemáticas, y la sección 1.5.4 ofrece una guía práctica.

Técnicas de diseño de algoritmos

Varias técnicas importantes de diseño de algoritmos vuelven a aparecer en muchos algoritmos. Ellas incluyen divide y vencerás, métodos codiciosos, búsqueda primero en profundidad (para grafos) y programación dinámica. Esta edición hace más hincapié que la segunda en las técnicas de diseño de algoritmos. La programación dinámica, igual que antes, tiene su propio capítulo, y la búsqueda primero en profundidad se presenta con muchas aplicaciones en el capítulo sobre recorrido de grafos (capítulo 7). Casi todos los capítulos están organizados por área de aplicación, no por técnica de diseño, por lo que a continuación presentaremos una lista de lugares en los que el lector hallará algoritmos que usan técnicas de divide y vencerás y codiciosas.

La técnica de dividir y vencer se describe en la sección 4.3 y se usa en la Búsqueda Binaria (sección 1.6), en casi todos los métodos de ordenamiento (capítulo 4), en la determinación de medianas y en el problema de selección general (sección 5.4), en los árboles de búsqueda binaria (sección 6.4), en la evaluación de polinomios (sección 12.2), en la multiplicación de matrices (sección 12.3), en la Transformada Rápida de Fourier (sección 12.4), en el coloreado aproximado de grafos (sección 13.7) y, en una forma un poco distinta, en la computación en paralelo en la sección 14.5.

Los algoritmos codiciosos se usan para hallar árboles abarcantes mínimos y caminos más cortos en el capítulo 8, y en varios algoritmos de aproximación para problemas de optimización $\mathcal{NP}$-completos, como llenado de cajones, mochila, coloreado de grafos y vendedor viajero (véanse las secciones 13.4 a 13.8).

Cambios respecto a la segunda edición

Esta edición tiene tres capítulos y muchos temas que son nuevos. En todo el libro se han vuelto a escribir numerosas secciones incorporando cambios extensos. Unos cuantos temas de la segunda edición se han pasado a otros capítulos donde, creemos, encajan mejor. Añadimos más de 100 ejercicios nuevos, muchas citas bibliográficas y un apéndice con ejemplos de Java. Los capítulos 2, 3 y 6 son prácticamente nuevos en su totalidad.

El capítulo 2 repasa los tipos de datos abstractos (TDA) e incluye especificaciones para varios TDA estándar. En todo el libro se hace hincapié en el papel de los tipos de datos abstractos en el diseño de algoritmos.

El capítulo 3 repasa la recursión y la inducción, haciendo hincapié en la conexión entre los dos y su utilidad en el diseño de programas y en la demostración de que son correctos. En este capítulo también se desarrollan los árboles de recursión, que proporcionan una representación visual e intuitiva de las ecuaciones de recurrencia que surgen durante el análisis de algoritmos recursivos. Las soluciones para patrones que se presentan a menudo se resumen con el fin de facilitar su uso en capítulos posteriores.

El capítulo 6 aborda el hashing o dispersión, los árboles rojinegros para árboles binarios equilibrados, las colas de prioridad avanzadas y las relaciones de equivalencia dinámica (Unión-Hallar). Este último tema se trataba en otro capítulo en la segunda edición.

Reescribimos todos los algoritmos en un pseudocódigo basado en Java. No es necesario saber Java; cualquiera que esté familiarizado con C o C++ podrá leer fácilmente los algoritmos. El capítulo 1 incluye una introducción al pseudocódigo basado en Java.

Hemos ampliado considerablemente la sección que trata las herramientas matemáticas para el análisis de algoritmos en el capítulo 1 con el fin de proporcionar un mejor repaso y una referencia de las matemáticas que se usan en el libro. El tratamiento del orden asintótico de las funciones de la sección 1.5 se diseñó pensando en ayudar a los estudiantes a dominar mejor los conceptos y técnicas relacionados con el orden asintótico. Añadimos reglas, en lenguaje informal, que resumen los casos más comunes (véase la sección 1.5.4).

El capítulo 4 contiene una versión acelerada de Heapsort en la que el número de comparaciones de claves se recorta casi a la mitad. En el caso de Quicksort, usamos el algoritmo de partición de Hoare en el texto principal. El método de Lomuto se introduce en un ejercicio. (En la segunda edición se hizo al revés.)

Hemos dividido el antiguo capítulo sobre grafos en dos, y cambiamos el orden de algunos temas. El capítulo 7 se concentra en los algoritmos de recorrido (en tiempo lineal). La presentación de la búsqueda primero en profundidad se modificó exhaustivamente destacando la estructura general de la técnica y mostrando más aplicaciones. Añadimos ordenamiento topológico y análisis de rutas críticas como aplicaciones, en vista de su valor intrínseco y su relación con la programación dinámica. Presentamos el algoritmo de Sharir, en vez del de Tarjan, para determinar componentes conectados.

El capítulo 8 trata los algoritmos codiciosos para problemas de grafos. Las presentaciones del algoritmo de Prim para árboles abarcantes mínimos y del algoritmo de Dijkstra para caminos más cortos se reescribieron tratando de hacer hincapié en el papel que desempeñan las colas de prioridad y de ilustrar la forma en que el uso de tipos de datos abstractos puede conducir al diseñador a implementaciones eficientes. Se menciona la implementación asintóticamente óptima $\Theta(m + n \log n)$, pero no se analiza a fondo. Trasladamos el algoritmo de Kruskal para árboles abarcantes mínimos a este capítulo.

La presentación de la programación dinámica (capítulo 10) se modificó sustancialmente a fin de hacer hincapié en un enfoque general para hallar soluciones de programación dinámica. Añadimos una nueva aplicación, un problema de formateo de texto, para subrayar el punto de que no todas las aplicaciones requieren un arreglo bidimensional. Pasamos la aplicación de cotejo aproximado de cadenas (que en la segunda edición estaba en este capítulo) al capítulo sobre cotejo de cadenas (sección 11.5). Los ejercicios incluyen otras aplicaciones nuevas.

Nuestra experiencia docente ha revelado áreas específicas en que los estudiantes tuvieron dificultad para captar conceptos relacionados con $\mathcal{P}$ y $\mathcal{NP}$ (capítulo 13), sobre todo algoritmos no deterministas y transformaciones polinómicas. Reescribimos algunas definiciones y ejemplos para dejar más claros los conceptos. Añadimos una sección corta sobre algoritmos de aproximación para el problema del vendedor viajero y una sección acerca de la computación por ADN.

A los profesores que usaron la segunda edición seguramente les interesará saber que modificamos algunas convenciones y términos (casi siempre para adecuarlos al uso común). Los arreglos de índices ahora principian en 0 en vez de 1 en muchos casos. (En otros, en los que la numeración a partir de 1 era más clara, la dejamos así.) Ahora usamos el término *profundidad* en lugar de *nivel* para referirnos a la ubicación "vertical" de un nodo dentro de un árbol. Usamos *altura* en vez de *profundidad* para referirnos a la profundidad máxima de cualquier nodo de un árbol. En la segunda edición, un *camino* en un grafo se definía como lo que comúnmente se conoce como *camino simple*; en esta edición usamos la definición más general de *camino* y definimos *camino simple* aparte. Ahora un grafo dirigido puede contener una auto-arista.

Ejercicios y programas

Algunos ejercicios son un tanto "abiertos". Por ejemplo, uno de ellos podría pedir una buena cota inferior para la complejidad de un problema, en lugar de pedir a los estudiantes demostrar que una función dada es una cota inferior. Hicimos esto por dos razones. Una fue hacer más realista el planteamiento de la pregunta; las soluciones se tienen que descubrir, no nada más verificarse. La otra es que para algunos estudiantes podría ser difícil demostrar la mejor cota inferior conocida (o hallar el algoritmo más eficiente para un problema), pero incluso en esos casos habrá una gama de soluciones que podrán ofrecer para demostrar su dominio de las técnicas estudiadas.

Algunos temas y problemas interesantes se introducen únicamente en los ejercicios. Por ejemplo, el problema del conjunto independiente máximo para un árbol es un ejercicio del capítulo 3, el problema de la sumatoria de subsucesión máxima es un ejercicio del capítulo 4, y el problema de hallar un sumidero para un grafo es un ejercicio del capítulo 7. Varios problemas $\mathcal{NP}$-completos se introducen en ejercicios del capítulo 13.

Las capacidades, antecedentes y conocimientos matemáticos de los estudiantes de diferentes universidades varían considerablemente, lo que dificulta decidir exactamente cuáles ejercicios deben marcarse (con un asterisco) como "difíciles". Marcamos los ejercicios que requieren matemáticas más allá de las básicas, los que requieren mucha creatividad y para los que se debe seguir una cadena de razonamiento larga. Unos cuantos ejercicios tienen dos asteriscos. Algunos ejercicios con asterisco tienen sugerencias.

Los algoritmos presentados en este libro no son programas; es decir, se han omitido muchos detalles que no son importantes para el método o para el análisis. Claro que los estudiantes deben saber cómo implementar algoritmos eficientes en programas eficientes sin errores. Muchos profesores podrían impartir este curso como curso "teórico" puro, sin programación. Para quienes desean asignar proyectos de programación, casi todos los capítulos incluyen una lista de tareas de programación: sugerencias breves que los profesores que decidan usarlas tal vez necesitarán ampliar.

Cómo seleccionar temas para un curso

Es evidente que la cantidad de material y la selección específica de los temas a cubrir dependerán del curso específico y de la población de estudiantes. Presentaremos cuadros sinópticos de ejemplo para dos cursos de licenciatura y uno de posgrado.

Este programa corresponde aproximadamente al curso para cuarto año (septimo u octavo semestre) que Sara Baase imparte en la San Diego State University durante un semestre de 15 semanas con 3 horas por semana de clase.

Capítulo 1: Se deja como lectura todo el capítulo, pero concentrándose en las secciones 1.4 y 1.5 en clase.
Capítulo 2: Las secciones 2.1 a 2.4 se dejan como lectura.
Capítulo 3: Las secciones 3.1 a 3.4, 3.6 y 3.7 se dejan como lectura, cubriéndose someramente en clase.
Capítulo 4: Secciones 4.1 a 4.9.
Capítulo 5: Secciones 5.1 a 5.2, 5.6 y parte de la 5.4.
Capítulo 7: Secciones 7.1 a 7.4 y 7.5 o bien 7.6, y 7.7.
Capítulo 8: Secciones 8.1 a 8.3, con breve mención de la 8.4.
Capítulo 11: Secciones 11.1 a 11.4.
Capítulo 13: Secciones 13.1 a 13.5, 13.8 y 13.9.

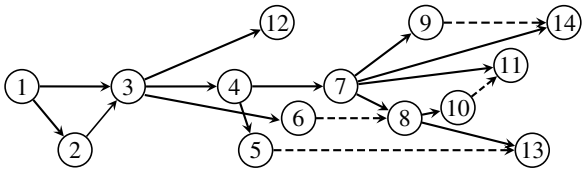
El programa que sigue es el curso de tercer año que Allen Van Gelder imparte en la University of California, Santa Cruz, en un trimestre de 10 semanas con 3.5 horas de clase por semana.

Capítulo 1: Secciones 1.3 y 1.5; las demás se dejan como lectura.
Capítulo 2: Secciones 2.1 a 2.3; las demás se dejan como lectura.
Capítulo 3: Se tocan todas las secciones; una buena parte se deja como lectura.
Capítulo 4: Secciones 4.1 a 4.9.
Capítulo 5: Posiblemente la sección 5.4, sólo el algoritmo en tiempo lineal promedio.
Capítulo 6: Secciones 6.4 a 6.6.
Capítulo 7: Secciones 7.1 a 7.6.
Capítulo 8: Todo el capítulo.
Capítulo 9: Secciones 9.1 a 9.4.
Capítulo 10: Posiblemente las secciones 10.1 a 10.3, pero casi nunca alcanza el tiempo.

Para el curso de primer año de posgrado de la University of California, Santa Cruz (también 10 semanas, 3.5 horas de clases), el material anterior se comprime y se cubren los temas adicionales siguientes:

Capítulo 5: Todo el capítulo.
Capítulo 6: El resto del capítulo, con hincapié en el análisis amortizado.
Capítulo 10: Todo el capítulo.
Capítulo 13: Secciones 13.1 a 13.3, y posiblemente la sección 13.9.

Las dependencias primarias entre los capítulos se muestran en el diagrama siguiente con líneas continuas; algunas dependencias secundarias de indican con líneas interrumpidas. Una dependencia secundaria implica que sólo se necesitan unos cuantos temas del capítulo anterior en el capítulo posterior, o que sólo las secciones más avanzadas del capítulo posterior requieren haber visto el anterior.



Aunque es importante haber visto el material de los capítulos 2 y 6, es posible que una buena parte de él ya se haya cubierto en un curso anterior. Algunas secciones del capítulo 6 son importantes para las partes más avanzadas del capítulo 8.

Nos gusta recordar a los lectores temas o técnicas comunes, por lo que abundan las referencias a secciones anteriores; se puede hacer caso omiso de muchas de esas referencias si no se cubrieron las secciones anteriores. Varios capítulos tienen una sección que trata cotas inferiores y puede ser provechoso haber visto antes las ideas y ejemplos del capítulo 5, pero el diagrama no muestra esa dependencia porque muchos profesores no tratan las cotas inferiores.

Hemos marcado (con un asterisco) las secciones que contienen matemáticas más complicadas o argumentos más complejos o avanzados que la generalidad de las otras, pero sólo en los casos en que el material no es fundamental para el libro. También marcamos una o dos secciones que contienen digresiones opcionales. No marcamos unas cuantas secciones que consideramos indispensables para un curso en el que se usa el libro, aunque contienen muchas matemáticas. Por ejemplo, es importante cubrir al menos una parte del material de la sección 1.5 sobre la tasa de crecimiento asintótico de las funciones y la sección 3.7 sobre soluciones de ecuaciones de recurrencia.

Agradecimientos

Nos alegramos de tener esta oportunidad para agradecer a las personas que nos ayudaron mucho o poco en la preparación de la tercera edición de este libro.

Sara Baase reconoce la influencia e inspiración de Dick Karp, quien hizo tan interesante y hermoso el tema de la complejidad computacional en sus soberbias conferencias. Allen Van Gelder hace patente su reconocimiento por las revelaciones que tuvo gracias a Bob Floyd, Don Knuth, Ernst Mayr, Vaughan Pratt y Jeff Ullman; todos ellos enseñan más de lo que está "en el libro". Allen también quiere expresar su reconocimiento a sus colegas David Helmbold, por muchas conversaciones acerca de la forma de presentar eficazmente los algoritmos y acerca de los puntos finos de muchos de ellos, y Charlie McDowell por su ayuda con muchos de los aspectos de Java que se cubren en el apéndice de esta obra. Agradecemos a Lila Kari haber leído uno de los primeros borradores de la sección acerca de computación por ADN y aclarar nuestras dudas.

Desde luego, no tendríamos nada acerca de qué escribir sin las muchas personas que realizaron las investigaciones originales de las cuales se deriva el material que nos gusta aprender y comunicar a nuevas generaciones de estudiantes. Estamos en deuda con ellos por su trabajo.

En los años que han pasado desde que apareció la segunda edición, varios estudiantes y profesores que usaron el libro enviaron listas de equivocaciones, errores tipográficos y sugerencias de cambios. No tenemos una lista completa de sus nombres, pero apreciamos el tiempo y la meditación que invirtieron en sus cartas.

Las encuestas y reseñas del manuscrito obtenidas por Addison Wesley fueron de especial utilidad. Muchas gracias a Iliana Bjorling Sachs (del Lafayette College), Mohammad B. Dadfar (Bowling Green State University), Daniel Hirschberg (University of California in Irvine), Mitsu-

nori Ogihara (University of Rochester), R. W. Robinson (University of Georgia), Yaakov L. Varol (University of Nevada, Reno), William W. White (Southern Illinois University at EdwardsVille), Dawn Wilkins (University of Mississippi) y Abdou Youssef (George Washington University).

Agradecemos a nuestras editoras de Addison Wesley, Maité Suárez-Rivas y Karen Wernholm, su confianza y paciencia al trabajar con nosotros en este proyecto que a menudo se apartó de los procedimientos y calendarios estándar de la producción. Muchas gracias a Joan Flaherty por su minuciosa edición y valiosas sugerencias para mejorar la presentación. Las cuidadosas lecturas de Brooke Albright detectaron muchos errores que habían sobrevivido a escrutinios anteriores; desde luego, todos los que queden son por culpa de los autores.

Agradecemos a Keith Mayers su ayuda en muchos sentidos. Sara le da las gracias por no haberle recordado con demasiada frecuencia que rompió la promesa que le hizo al casarse de trabajar menos de siete días a la semana.

Sara Baase, *San Diego, California*
http://www-rohan.sdsu.edu/faculty/baase

Allen Van Gelder, *Santa Cruz, California*
http://www.cse.ucsc.edu/personnel/faculty/avg.html

Junio de 1999

Contenido

1

Análisis de algoritmos y problemas: principios y ejemplos

1.1 Introducción

Decir que un problema se puede resolver algorítmicamente implica, informalmente, que es posible escribir un programa de computadora que producirá la respuesta correcta para cualquier entrada si permitimos que se ejecute durante el tiempo suficiente y le proporcionamos todo el espacio de almacenamiento que necesite. En la década de 1930, antes de la llegada de las computadoras, los matemáticos trabajaron con gran celo para formalizar y estudiar el concepto de algoritmo, que entonces se definía de manera informal como un conjunto claramente especificado de instrucciones sencillas a seguir para resolver un problema o calcular una función. Se idearon e investigaron varios modelos de cómputo formales. Muchos de los primeros trabajos en este campo, llamado *teoría de la computabilidad*, hicieron hincapié en describir o caracterizar los problemas que se podían resolver algorítmicamente, y en presentar algunos problemas que no se podían resolver de esa manera. Uno de los resultados negativos importantes, establecido por Alan Turing, fue la demostración de la insolubilidad del "problema del paro". Este problema consiste en determinar si cualquier algoritmo (o programa de computadora) determinado llegará en algún momento a un punto en el que se detendrá (en vez de, digamos, entrar en un ciclo infinito) al trabajar con una entrada dada. No puede existir un programa de computadora que resuelva este problema.

Aunque la teoría de la computabilidad tiene implicaciones obvias y fundamentales para las ciencias de la computación, el saber que en teoría un problema se puede resolver con una computadora no basta para decirnos si resulta práctico hacerlo o no. Por ejemplo, se podría escribir un programa perfecto para jugar ajedrez. La tarea no sería demasiado difícil; las formas de acomodar las piezas de ajedrez en el tablero son finitas, y bajo ciertas reglas una partida debe terminar después de un número finito de movimientos. El programa podría considerar todos los movimientos posibles que podría efectuar la computadora, cada una de las posibles respuestas del oponente, cada una de sus posibles respuestas a esos movimientos, y así hasta que cada una de las sucesiones de posibles movimientos llegara a su fin. Entonces, dado que la computadora conoce el resultado final de cada movimiento, podrá escoger la mejor. Según algunas estimaciones, el número de acomodos distintos de las piezas en el tablero que es razonable considerar (mucho menor que el número de sucesiones de movimientos) es de aproximadamente 10^{50}. Un programa que las examinara todas tardaría varios miles de años en ejecutarse. Es por ello que no se ha ejecutado un programa semejante.

Es posible resolver una gran cantidad de problemas con aplicaciones prácticas —es decir, se pueden escribir programas para hacerlo— pero las necesidades de tiempo y almacenamiento son demasiado grandes para que tales programas tengan utilidad práctica. Es evidente que las necesidades de tiempo y espacio de los programas son importantes en la práctica; por ello, se han convertido en el tema de estudios teóricos en el área de las ciencias de la computación llamada *complejidad computacional*. Una rama de tales estudios, que no se cubrirá en este libro, se ocupa de establecer una teoría formal y un tanto abstracta de la complejidad de las funciones computables. (Resolver un problema equivale a calcular una función que a partir del conjunto de entradas proporcione el conjunto de salidas.) Se han formulado axiomas para medir la complejidad, éstos son básicos y lo bastante generales como para poder usar el número de instrucciones ejecutadas o bien el número de bits de almacenamiento que ocupa un programa como medida de su complejidad. Utilizado esos axiomas, podemos demostrar la existencia de problemas arbitrariamente complejos y de problemas para los que no existe un programa óptimo.

La rama de la complejidad computacional que estudiaremos en este libro se ocupa de analizar problemas específicos y algoritmos específicos. El libro pretende ayudar a los lectores a for-

mar un repertorio de algoritmos clásicos para resolver problemas comunes, algunas técnicas, herramientas y principios de diseño generales para analizar algoritmos y problemas, y métodos para demostrar que la solución es correcta. Presentaremos, estudiaremos y analizaremos algoritmos para resolver diversos problemas para los que comúnmente se usan programas de computadora. Analizaremos el tiempo que tardan en ejecutarse los algoritmos, y muchas veces también el espacio que consumen. Al describir algoritmos para diversos problemas, veremos que hay varias técnicas de diseño de algoritmos que a menudo resultan útiles. Por ello, haremos ahora una pausa para hablar acerca de algunas técnicas generales, como divide y vencerás, algoritmos codiciosos, búsqueda de primero en profundidad y programación dinámica. También estudiaremos la complejidad computacional de los problemas mismos, es decir, el tiempo y espacio que se requieren inherentemente para resolver el problema, sea cual sea el algoritmo empleado. Estudiaremos la clase de problemas $\mathcal{NP}$-completos —problemas para los que no se conocen algoritmos eficientes— y consideraremos algunas heurísticas para obtener resultados útiles. También describiremos un enfoque para resolver estos problemas empleando ADN en lugar de computadoras electrónicas. Por último, presentaremos el tema de los algoritmos para computadoras paralelas.

En las secciones que siguen bosquejaremos el lenguaje algorítmico, repasaremos algunos antecedentes y herramientas que se usarán en todo el libro, e ilustraremos los principales conceptos que intervienen en el análisis de algoritmos.

1.2 Java como lenguaje algorítmico

Escogimos Java como lenguaje algorítmico para este libro sopesando varios criterios. Los algoritmos deben ser fáciles de leer. Queremos concentrarnos en la estrategia y las técnicas de un algoritmo, no declaraciones y detalles de sintaxis que interesan al compilador. El lenguaje debe manejar abstracción de datos y descomposición de problemas, a fin de facilitar la expresión clara de ideas algorítmicas. El lenguaje debe ofrecer un camino práctico hacia la implementación; debe estar ampliamente disponible e incluir apoyo para el desarrollo de programas. La implementación y ejecución reales de algoritmos puede mejorar considerablemente la comprensión del estudiante, y no debe convertirse en una frustrante batalla con el compilador y el depurador. Por último, dado que este libro enseña algoritmos, no un lenguaje de programación, debe ser razonablemente fácil traducir un algoritmo a diversos lenguajes que los lectores tal vez prefieran usar, por lo que conviene reducir al mínimo las características especializadas del lenguaje.

Java obtiene buenas calificaciones según varios de nuestros criterios, aunque no nos atreveríamos a decir que es ideal. Java apoya de forma natural la abstracción de datos; es seguro en cuanto a los tipos, lo que significa que objetos de un tipo no se pueden usar en operaciones diseñadas para un tipo distinto; tampoco se permiten conversiones arbitrarias de tipo (llamadas "casts", o "mutaciones"). Hay un tipo **`boolean`** explícito, de modo que si uno escribe "=" (el operador de asignación) cuando la intención era escribir "==" (el operador de igualdad), el compilador lo detecta.

Java no permite la manipulación de apuntadores, lo que a menudo dan pie a errores difíciles de encontrar; de hecho, los apuntadores están ocultos del programador y se manejan automáticamente tras bambalinas. En el momento de la ejecución, Java verifica que los subíndices de arreglos estén dentro del intervalo definido, y cuida que no haya otras incongruencias que podrían originar errores escondidos. Se efectúa "recolección de basura", o sea que se recicla el espacio de almacenamiento de objetos a los que ya no se hace referencia; esto alivia considerablemente la carga del programador en cuanto a administración del espacio.

En el lado negativo, Java tiene muchas de las parcas y misteriosas características de la sintaxis de C. La estructura de los objetos podría obligar a hacer uso ineficiente del tiempo y el espacio. Muchas construcciones de Java requieren escribir más que en otros lenguajes, como C.

Aunque Java tiene muchos recursos especializados, los algoritmos que presentamos en este libro en su mayor parte los evitan, en aras de la independencia respecto al lenguaje. De hecho, algunos pasos de un algoritmo podrían estar planteados en pseudocódigo para hacerlos más comprensibles. En esta sección describiremos un pequeño subconjunto de Java que se usará en el libro, así como las convenciones de pseudocódigo que empleamos para hacer más comprensibles los algoritmos. El apéndice A, específico para Java, proporciona algunos detalles de implementación adicionales para los lectores que desean escribir un programa funcional en Java, pero tales detalles no son necesarios para entender el grueso del texto.

1.2.1 Un subconjunto práctico de Java

No es importante tener un conocimiento exhaustivo de Java para entender los algoritmos del presente texto. En esta sección se presenta una perspectiva somera de las características de Java que sí aparecen, para aquellos lectores que deseen seguir de cerca los detalles de implementación. En algunos casos mencionamos recursos orientados a objetos de Java que se podrían usar, pero que evitamos con el fin de que el texto sea relativamente independiente del lenguaje; esto se hizo pensando principalmente en los lectores que manejan algún otro lenguaje orientado a objetos, como C++, pero que no están perfectamente familiarizados con Java. En el apéndice A se presenta un ejemplo de "programa principal" en Java. Existen muchos libros que cubren a fondo el lenguaje.

Los lectores que conocen bien Java sin duda notarán muchos casos en los que se podría haber usado algún bonito recurso de Java. Sin embargo, los *conceptos* en los que se basan los algoritmos no requieren recursos especiales, y queremos que dichos conceptos sean fáciles de captar y aplicar en diversos lenguajes, así que dejamos que sea el lector quien, habiendo captado los conceptos, adapte las implementaciones a su lenguaje favorito.

Los lectores que conocen la sintaxis de C reconocerán muchas similitudes en la sintaxis de Java: los bloques se delimitan con llaves, "{" y "}"; los índices de arreglo se encierran en corchetes, "[" y "]". Al igual que en C y C++, un arreglo bidimensional es en realidad un arreglo unidimensional cuyos elementos son a su vez arreglos unidimensionales, así que se necesitan dos pares de corchetes para acceder a un elemento, como en "`matriz[i][j]`". Los operadores "==", "!=", "<=" y ">=" son las versiones de teclado de los operadores de relación matemáticos "=", "≠", "≤" y "≥", respectivamente. En pseudocódigo, normalmente se usan de preferencia las versiones matemáticas. En los ejemplos del texto se usan los operadores "++" y "--" para incrementar y decrementar, pero nunca se usan incrustados en otras expresiones. También están los operadores "+=", "−=", "*=" y "/=" adoptados de C. Por ejemplo,

```
p += q;   /* Sumar q a p. */
y -= x;   //Restar x de y.
```

Como acabamos de ilustrar, los comentarios se extienden desde "//" hasta el fin de la línea, o desde "/*" hasta "*/", igual que en C++.

Las cabeceras de función normalmente tienen el mismo aspecto en Java y en C. La cabecera especifica la *rúbrica de tipo de parámetros* entre paréntesis después del nombre de la función, y especifica el *tipo devuelto* antes del nombre de la función. La combinación de tipo devuelto y rúbrica de tipo de parámetros se denomina *rúbrica de tipo completa*, o *prototipo*, de la función. Así,

```
int obtMin(ColaPrioridad cp)
```

nos dice que la función `obtMin` recibe un parámetro del tipo (o **`clase`**) `ColaPrioridad` y devuelve un resultado de tipo **`int`**.

Java tiene unos cuantos *tipos primitivos*, y todos los demás tipos se llaman *clases*. Los tipos primitivos son lógicos (**`boolean`**) y numéricos (**`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`** y **`double`**). En Java todas las clases (tipos no primitivos) son *de referencia*. Tras bambalinas, las variables declaradas en clases son "apuntadores"; sus valores son direcciones. Los ejemplares de clases se llaman *objetos*. El hecho de declarar una variable no crea un objeto. En general, los objetos se crean con un operador "**`new`**", que devuelve una referencia al nuevo objeto.

Los campos de datos de un objeto se denominan *campos de ejemplar* en la terminología de orientación a objetos. Se usa el operador binario punto para acceder a los campos de ejemplar de un objeto.

Ejemplo 1.1 Cómo crear y acceder a objetos en Java

Para este ejemplo, supóngase que la información de fecha tiene la siguiente estructura lógica anidada:

- `año`
  - `numero`
  - `esBisiesto`
- `mes`
- `dia`

Es decir, empleando terminología formal, `año` es un atributo compuesto que consiste en el atributo booleano `esBisiesto` y el atributo entero `numero`, mientras que `mes` y `dia` son atributos enteros simples. A fin de reflejar la estructura anidada, es preciso definir dos clases en Java, una para toda la fecha y otra para el campo `año`. Supóngase que escogemos los nombres `Fecha` y `Año`, respectivamente, para estas clases. Entonces declararíamos `numero` y `esBisiesto` como campos de ejemplar en la clase `Año` y declararíamos `año`, `mes` y `dia` como campos de ejemplar en la clase `Fecha`. Además, lo más probable es que definiríamos a `Año` como clase interna de `Fecha`. La sintaxis se muestra en la figura 1.1.

```
class Fecha
    {
    public Año año;
    public int mes;
    public int dia;

    public static class Año
       {
       public int numero;
       public boolean esBisiesto;
       }
    }
```

Figura 1.1 Sintaxis de Java para la clase `Fecha` con una clase interna `Año`

Sin la palabra clave **public**, no se podría acceder a los campos de ejemplar fuera de las clases Fecha y Año; por sencillez, los hacemos **public** aquí. Declaramos la clase interna Año como **static** para poder crear un ejemplar de Año que no esté asociado a algún objeto Fecha en particular. En este libro todas las clases internas serán **static**.

Supóngase que hemos creado un objeto Fecha al que hace referencia la variable fechaPago. Para acceder al campo de ejemplar año de este objeto usamos el operador punto, como en "fechaPago.año". Si el campo de ejemplar está en una clase (en lugar de estar en un tipo primitivo), operadores punto adicionales accederán a sus campos de ejemplar, como en "fechaPago.año.esBisiesto".

El enunciado de asignación copia sólo la *referencia*, o *dirección*, de un objeto de una clase; no crea una copia de los campos de ejemplar. Por ejemplo, "fechaAviso = fechaPago" hace que la variable fechaAviso *se refiera al mismo objeto que* la variable fechaPago. Por tanto, el fragmento de código siguiente probablemente sería un error lógico:

```
fechaAviso = fechaPago;
fechaAviso.dia = fechaPago.dia − 7;
```

En la sección 1.2.2 se amplía la explicación. ■

Los enunciados de control **if**, **else**, **while**, **for** y **break** tienen el mismo significado en Java que en C (y en C++) y se usan en este libro. Hay varios otros enunciados de control, pero no los usaremos. Las sintaxis de **while** y **for** son

while (condición para continuar **)** cuerpo
for (inicializador **;** condición para continuar **;** incremento **)** cuerpo

donde "inicializador" e "incremento" son enunciados sencillos (sin "{, }"), "cuerpo" es un enunciado cualquiera y "condición para continuar" es una expresión **booleana**. El enunciado **break** hace que el programa salga de inmediato del ciclo **for** o **while** circundante más cercano.[1]

Todas las clases forman un árbol (también llamado jerarquía) cuya raíz es la clase **Object**. Al declarar una clase nueva, es posible decir que *extiende* una clase previamente definida, y la nueva clase se convierte en hija de la clase antes definida dentro del árbol de clases. No crearemos este tipo de estructuras aquí, a fin de mantener el código lo más independiente del lenguaje que sea posible; sin embargo, se dan unos cuantos ejemplos en el apéndice A. Si la clase nueva no se declara de modo que extienda alguna clase, extiende **Object** por omisión. No necesitaremos estructuras de clase complejas para los algoritmos que se estudian en este texto.

Las operaciones con objetos se llaman *métodos* en la terminología de orientación a objetos; sin embargo, nos limitaremos al uso de *métodos estáticos*, que no son más que procedimientos y funciones. En nuestra terminología, *procedimiento* es una sucesión de pasos de cómputo que tiene un nombre y que puede invocarse (con parámetros); una *función* es un procedimiento que además devuelve un valor al invocador. En Java, un procedimiento que no devuelve valor alguno se declara con tipo devuelto **void**; C y C++ son similares en este sentido. El término *estático* es terminología técnica de Java, y significa que el método se puede aplicar a cualquier objeto u objetos de los tipos apropiados (el tipo de un objeto es su clase), en congruencia con la rúbrica de tipo del método (conocida como su prototipo). Un método estático no está "atado" a un objeto es-

[1] También sale de **switch**, pero este **término** no se utiliza en el libro.

pecífico. Los métodos estáticos se comportan como las funciones y procedimientos ordinarios de lenguajes de programación como C, Pascal, etc. Sin embargo, es preciso anteponer a su nombre la clase en la que se definieron, como en "`Lista.primero(x)`" para aplicar al parámetro `x` el método `primero` que se definió en la clase `Lista`.

En Java, los campos de ejemplar de un objeto son privados por omisión, lo que implica que sólo los métodos (funciones y procedimientos) definidos dentro de la misma clase pueden tener acceso a ellos. Esto es congruente con el tema del diseño de tipos de datos abstractos (TDA) de que sólo debe accederse a los objetos a través de las operaciones definidas para el TDA. El código que implementa estas operaciones de TDA (o métodos estáticos, o funciones y procedimientos) existe dentro de la clase y tiene conocimiento de los campos de ejemplar privados y de sus tipos. Los métodos también son privados por omisión, pero casi siempre se especifican como "públicos" para que métodos definidos en otras clases puedan invocarlos. No obstante, los métodos "de bajo nivel" que sólo deben ser invocados por otros métodos de la misma clase bien podrían ser privados.

Los *clientes* del TDA (procedimientos y funciones que invocan el TDA) se implementan fuera de la clase en la que "vive" el TDA, de manera que sólo tienen acceso a las partes *públicas* de la clase de TDA. El mantenimiento de datos privados se denomina *encapsulamiento*, u *ocultamiento de información*.

Los campos de ejemplar de un objeto conservan los valores que se les asignan mientras existe el objeto, o hasta que una asignación posterior los sobreescribe. Aquí es evidente la ventaja de hacer que sean privados respecto a la clase en la que se definen. Cualquier parte del programa general podría asignar un valor arbitrario a un campo de ejemplar público. En cambio, sólo es posible asignar un valor a un campo de ejemplar privado utilizando un método de la clase de TDA diseñado para ese fin. Dicho método podría efectuar otros cálculos y pruebas para cerciorarse de que el valor asignado a un campo de ejemplar sea congruente con las especificaciones del TDA, y con los valores almacenados en otros campos de ejemplar del mismo objeto.

Se crea un objeto nuevo con la frase "**`new`** `nombreClase()`"; por ejemplo:

```
Fecha fechaPago = new Fecha();
```

Este enunciado hace que Java invoque un *constructor* por omisión para la clase `Fecha`. El constructor reserva espacio para un objeto (o ejemplar) nuevo de la clase y devuelve una referencia (probablemente una dirección) para acceder a ese objeto. Los campos de ejemplar de ese nuevo objeto podrían estar inicializados (es decir, tener valores iniciales) o no.

Detalle de Java: El programador podría escribir funciones constructoras adicionales para una clase, cuyos cuerpos podrían inicializar diversos campos de ejemplar y realizar otros cálculos. En aras de la independencia respecto al lenguaje, no usaremos aquí tales constructores, por lo que omitiremos los pormenores.

Los arreglos se declaran de forma un poco diferente en Java que en C y C++, y sus propiedades también presentan algunas diferencias. La sintaxis de Java para declarar un arreglo de enteros (o, en términos más precisos, para declarar una variable cuyo tipo es "arreglo de enteros") es "**`int`**`[] x`", mientras que en C se podría usar "**`int`** `x[]`". Este enunciado no inicializa a `x`; eso se hace con

```
x = new int[cuantos];
```

donde `cuantos` es una constante o una variable cuyo valor denota la longitud deseada para el arreglo. Las declaraciones de arreglos de clases son similares. La declaración y la inicialización pueden, y por lo regular deberían, combinarse en un solo enunciado:

```
int[] x = new int[cuantos];
Fecha[] fechas = new Fecha[cuantas];
```

Aunque estos enunciados inicializan a `x` y `fechas` en el sentido de que reservan espacio para los arreglos, sólo asignan a los *elementos* valores por omisión, que con toda seguridad no serán útiles. Por ello, es preciso asignar valores a los elementos individuales `fechas[0]`, `fechas[1]`,..., (posiblemente utilizando el operador **new**) antes de usarlos. La sintaxis, fuera de la clase `Fecha`, es

```
fechas[0] = new Fecha();
fechas[0].mes = 1;
fechas[0].dia = 1;
fechas[0].año = new Fecha.Año();
fechas[0].año.numero = 2000;
fechas[0].año.esBisiesto = true;
```

Observe que los nombres de campo van después del índice que selecciona un elemento específico del arreglo. Observe también que el nombre de la clase interior, `Año`, está calificado por el nombre de la clase exterior, `Fecha`, en el segundo enunciado **new**, porque el enunciado está afuera de la clase `Fecha`. Como ya se dijo, quienes programan en Java pueden escribir constructores que reciban parámetros para efectuar este tipo de inicialización de objetos recién construidos, pero en el presente texto no usaremos tales constructores en aras de la independencia respecto al lenguaje.

Una vez inicializado el arreglo `x` con un enunciado **new**, como se mostró unos párrafos atrás, ya no se podrá modificar la longitud del arreglo al que hace referencia. Java ofrece un mecanismo para consultar esa longitud, que es `x.length`. Es decir, el *campo de ejemplar* `length` se "anexa" automáticamente al objeto de arreglo como parte de la operación **new**, y se puede acceder a él a través de `x`, como se muestra, en tanto `x` se refiera a ese objeto.

Los índices (o subíndices) válidos para elementos de este arreglo son del 0 a (`x.length` − 1). Java detendrá el programa (en términos técnicos, lanzará una excepción) si éste intenta acceder a un elemento cuyo índice está fuera de ese intervalo. Con frecuencia queremos usar índices dentro del intervalo de 1 a *n*, así que inicializaremos los arreglos con "`new int[n+1]`" en esos casos.

Java permite *sobrecargar* y *suplantar* métodos. Decimos que un método está *sobrecargado* si tiene dos o más definiciones con distintos tipos de parámetros, pero su tipo devuelto es el mismo. Muchos operadores aritméticos están sobrecargados. *Suplantar* implica que en la jerarquía de clases hay varias definiciones de un mismo método, con los mismos tipos de parámetros, y que Java aplica la definición "más cercana". (Una vez más, por compatibilidad con otros lenguajes y porque esta capacidad no es fundamental para entender los algoritmos, evitaremos estos recursos y remitiremos al lector interesado a libros que tratan el lenguaje Java.) Se pueden usar los mismos nombres de métodos en diferentes clases, pero esto no es realmente sobrecargar porque el nombre de clase (o de objeto) aparece como calificador cuando los nombres se usan fuera de la clase en la que se definen. Esto se aclarará con ejemplos posteriores.

Para los lectores que conocen C++, vale la pena señalar que Java no permite al programador definir significados nuevos para los *operadores*. En este texto usamos tales operadores en

pseudocódigo para hacerlo más legible (por ejemplo, $x < y$, donde x y y pertenecen a alguna clase no numérica, como **`String`**). No obstante, si usted define una clase y escribe un programa real en Java que la usa, deberá escribir funciones con nombre (por ejemplo `menor()`) e invocarla para comparar objetos de esa clase.

1.2.2 Clases organizadoras

Acuñamos el término *clase organizadora*, que no es un término estándar de Java, para describir una clase muy sencilla que simplemente agrupa varios campos de ejemplar. Esta construcción desempeña un papel hasta cierto punto análogo al de *struct* en C y *record* en Pascal o Modula; existen construcciones análogas en Lisp, ML, y casi todos los demás lenguajes de programación. Las clases organizadoras tienen un propósito diametralmente opuesto al de los tipos de datos abstractos; se limitan a organizar una porción de almacenamiento, pero no limitan el acceso a él ni proporcionan operaciones a la medida para él. En muchos casos es conveniente definir una clase organizadora dentro de alguna otra clase; en este caso, la clase organizadora sería una *clase interna* en la terminología de Java.

Una clase organizadora sólo tiene un método, llamado `copy`. Puesto que las variables son *referencias* a objetos en Java, el enunciado de asignación sólo copia la referencia, no los campos del objeto, como se ilustró en el ejemplo 1.1 con `fechaPago` y `fechaAviso`. Si estas variables se declaran en una clase organizadora llamada `Fecha`, podríamos usar los enunciados

```
fechaAviso = Fecha.copy(fechaPago);
fechaAviso.dia = fechaPago.dia - 7;
```

para copiar los campos de `fechaPago` en un objeto nuevo al que hace referencia `fechaAviso`, y luego modificar únicamente el campo `dia` de `fechaAviso`.

Definición 1.1 La función `copy` de clases organizadoras

La regla general para la forma en que la función (o método) `copy` de una clase organizadora debe asignar valores a los campos de ejemplar del nuevo objeto (que se ilustrará suponiendo que el objeto `f` se está copiando en un objeto nuevo `f2`) es la siguiente:

1. Si el campo de ejemplar (digamos `año`) está en otra clase *organizadora*, entonces se invoca el método `copy` de esa clase, como en `f2.año = Año.copy(f.año)`.
2. Si el campo de ejemplar (digamos `dia`) *no* está en una clase *organizadora*, se usará una asignación simple, como en `f2.dia = f.dia`.

El ejemplo completo se muestra en la figura 1.2. ■

El programador debe cuidar que no haya ciclos en las definiciones de las clases organizadoras, pues si los hubiera es posible que `copy` nunca termine. Desde luego, también se puede crear un objeto nuevo en una clase organizadora de la forma acostumbrada:

```
Fecha algunaFecha = new Fecha();
```

Detalle de Java: Java cuenta con un recurso, basado en el método **`clone`**, para crear una copia de un solo nivel de un objeto sin tener que escribir todos y cada uno de los enunciados de asignación, pero dicho recurso no maneja automáticamente estructuras anidadas como `Fecha`; será necesario escribir algo de código en esos casos. En el apéndice A se da el código para una función `copiar1nivel` "genérica".

```
class Fecha
    {
    public Año año;
    public int mes;
    public int dia;

    public static class Año
       {
       public int numero;
       public boolean esBisiesto;

       public static Año copy(Año a)
         { Año a2 = new Año();
           a2.numero = a.numero;
           a2.esBisiesto = a.esBisiesto;
           return a2;
         }
       }
    public static Fecha copy(Fecha f)
       { Fecha f2 = new Fecha();
         f2.año = Año.copy(f.año);  // clase organizadora
         f2.mes = f.mes;
         f2.dia = f.dia;
         return f2;
       }
    public static int sigloPorOmision;
    }
```

Figura 1.2 Una clase organizadora `Fecha` con una clase principal organizadora de `Año`

Una clase organizadora sólo contiene campos de ejemplar **`public`**. Si también aparece la palabra clave **`static`** en la declaración del campo, el campo no estará asociado a ningún objeto específico, y será básicamente una variable global.

Ejemplo 1.2 Clases organizadoras típicas

En la figura 1.2 se adornan las clases del ejemplo 1.1 con funciones `copy`, para que sean clases organizadoras. Como se ve, la definición de `copy` es mecánica, aunque tediosa. Sus detalles se omitirán en ejemplos futuros. Para que el ejemplo esté completo, incluimos `sigloPorOmision` como ejemplo de "variable global", aunque la mayor parte de las clases organizadoras no contendrá variables globales. ■

En síntesis, inventamos el término *clase organizadora* para denotar una clase que simplemente agrupa algunos campos de ejemplar y define una función para crear copias de los mismos.

1.2.3 Convenciones de pseudocódigo basado en Java

La mayor parte de los algoritmos de este libro utilizan un pseudocódigo basado en Java, en lugar de Java estricto, para facilitar la comprensión. Se aplican las convenciones siguientes (excepto en el apéndice A, específico para Java).

1. Se omiten los delimitadores de bloques ("{" y "}"). Los límites de los bloques se indican con sangrías.
2. Se omite la palabra clave **`static`** en las declaraciones de métodos (funciones y procedimientos). Todos los métodos declarados en el texto son **`static`**. (De vez en cuando aparecen métodos no estáticos que vienen incluidos en Java; en particular, se usa `c.length()` para obtener la longitud de una cadena.) La palabra clave **`static`** *sí* aparece cuando se necesita para definir campos de ejemplar y clases internas.
3. Se omiten los calificadores de nombre de clase en las invocaciones de métodos (funciones y procedimientos). Por ejemplo, se podría escribir `x = cons(z, x)` cuando la sintaxis de Java exige `x = ListaInt.cons(z, x)`. (La clase `ListaInt` se describe en la sección 2.3.2.) Los calificadores de nombre de clase son obligatorios en Java siempre que se invocan métodos estáticos desde afuera de la clase en la que se definen.
4. Se omiten las palabras clave que sirven para controlar la visibilidad: **`public`**, **`private`** y **`protected`**. Si se colocan todos los archivos relacionados con un programa Java dado en el mismo directorio se hace innecesario ocuparse de cuestiones de visibilidad.
5. Por lo regular se escriben los operadores de relación matemáticos "≠", "≤" y "≥", en lugar de sus versiones de teclado. Se usan operadores de relación con tipos para los cuales el significado es obvio, como **`String`**, aunque esto no lo permitiría la sintaxis de Java.
6. Las palabras clave, que son palabras reservadas o bien componentes estándar de Java, aparecen en este tipo de letra: **`int`**, **`String`**. *`Los comentarios están en este tipo de letra.`* `Los enunciados de código y los nombres de variables de programa aparecen en este tipo de letra.` En cambio, los enunciados de pseudocódigo se escriben empleando el tipo de letra normal del texto, como esta oración.

Ocasionalmente nos apartaremos de este esquema para destacar algún aspecto específico del lenguaje Java.

1.3 Antecedentes matemáticos

Utilizamos diversos conceptos, herramientas y técnicas matemáticas en este libro. En su mayor parte, el lector ya las conocerá, aunque unas cuantas podrían serle nuevas. En esta sección se reúnen para poder consultarlas fácilmente, y hacer un repaso somero. Los conceptos de las demostraciones se cubren más a fondo en el capítulo 3.

1.3.1 Conjuntos, tuplas y relaciones

En esta sección presentamos definiciones informales y unas cuantas propiedades elementales de los conjuntos y conceptos afines. Un conjunto es una colección de elementos distintos que queremos tratar como un solo objeto. Por lo regular los objetos son del mismo "tipo" y tienen en co-

mún algunas otras propiedades que hacen que sea útil pensar en ellos como un solo objeto. La notación $e \in S$ se lee "el elemento e es un miembro del conjunto S" o, más brevemente, "e está en S". Cabe señalar que en este caso e y S son de diferente tipo. Por ejemplo, si e es un entero, S es un *conjunto de* enteros, que no es lo mismo que ser un entero.

Un conjunto dado se define enumerando o describiendo sus elementos entre un par de llaves. He aquí algunos ejemplos de esta notación:

$$S_1 = \{a, b, c\}, \qquad S_2 = \{x \mid x \text{ es una potencia entera de } 2\}, \qquad S_3 = \{1, \ldots, n\}.$$

La expresión para S_2 se lee "el conjunto de *todos* los elementos x *tales que* x es una potencia entera de 2". El símbolo "|" se lee "tales que" en este contexto. A veces se usa un signo de dos puntos (":") en vez de "|". Se pueden usar puntos suspensivos "…" cuando es obvio cuáles son los elementos implícitos.

Si todos los elementos de un conjunto S_1 también están en otro conjunto S_2, decimos que S_1 es un *subconjunto* de S_2 y que S_2 es un *superconjunto* de S_1. Las notaciones son $S_1 \subseteq S_2$ y $S_2 \supseteq S_1$. Para denotar que S_1 es un subconjunto de S_2 *y no es igual a* S_2, escribimos $S_1 \subset S_2$ o $S_2 \supset S_1$. Es importante no confundir "$\in$" con "$\subset$". El primero significa "es un elemento de" y el segundo implica "es un conjunto de elementos contenido en". El *conjunto vacío*, denotado por $\emptyset$, no tiene elementos, así que es un subconjunto de todos los conjuntos.

Un conjunto no tiene un orden inherente. Así pues, en los ejemplos anteriores, podríamos haber definido a S_1 como $\{b, c, a\}$ y S_3 podría haberse definido como $\{i \mid 1 \leq i \leq n\}$ si se entiende que i es un entero.

Un grupo de elementos que está *en un orden específico* se denomina *sucesión*. Además del orden, otra diferencia importante entre los conjuntos y las sucesiones es que las sucesiones pueden tener elementos repetidos. Las sucesiones se denotan enumerando sus elementos en orden, encerrados en paréntesis. Así, (a, b, c), (b, c, a) y (a, b, c, a) son sucesiones distintas. También se pueden usar puntos suspensivos en las sucesiones, como en $(1, \ldots, n)$.

Un conjunto S es *finito* si hay un entero n tal que los elementos de S se puedan colocar en una correspondencia uno a uno con $\{1, \ldots, n\}$; en este caso escribimos $|S| = n$. En general, $|S|$ denota el número de elementos que hay en el conjunto S, y también se denomina *cardinalidad* de S. Una sucesión es *finita* si existe un entero n tal que los elementos de la sucesión se puedan colocar en una correspondencia uno a uno con $(1, \ldots, n)$. Un conjunto o sucesión que no es finito es *infinito*. Si todos los elementos de una sucesión finita son distintos, decimos que esa sucesión es una *permutación* del *conjunto* finito que consta de los mismos elementos. Esto destaca una vez más la diferencia entre un conjunto y una sucesión. Un conjunto de n elementos tiene $n!$ permutaciones distintas (véase la sección 1.3.2).

¿Cuántos subconjuntos distintos tiene un conjunto finito de n elementos? Tenga presente que el conjunto vacío y el conjunto total son subconjuntos. Para construir cualquier subconjunto tenemos n decisiones binarias: incluir o excluir cada elemento del conjunto dado. Hay 2^n formas distintas de tomar esas decisiones, así que hay 2^n subconjuntos.

¿Cuántos subconjuntos distintos *con cardinalidad* k tiene un conjunto finito de n elementos? Existe una notación especial para esta cantidad: $\binom{n}{k}$, que se lee "n selecciones de k", o de forma más explícita, "número de combinaciones de n cosas tomadas k a la vez". También se usa la notación $C(n, k)$, y estas cantidades se denominan *coeficientes binomiales*.

Si queremos obtener una expresión para $\binom{n}{k}$ o $C(n, k)$, nos concentramos en las opciones en el subconjunto de k en lugar de las opciones en el conjunto original, digamos S. Podemos obtener

una *sucesión* de k elementos distintos de S como sigue: como primer elemento de la sucesión se puede escoger cualquier elemento de S, así que hay n opciones. Luego, como segundo elemento de la sucesión se puede escoger cualquier elemento restante de S, así que hay $(n - 1)$ opciones en este caso, y así hasta escoger k elementos. (Si $k > n$ es imposible escoger k opciones distintas, así que el resultado es 0.) Por tanto, hay $n(n - 1) \cdots (n - k + 1)$ sucesiones distintas de k elementos distintos. Pero vimos que un conjunto específico de k elementos se puede representar como $k!$ sucesiones. Entonces, el número de subconjuntos distintos de k, tomados de un conjunto de n, es

$$C(n, k) \equiv \binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!} = \frac{n!}{(n-k)!k!} \qquad \text{para } n \geq k \geq 0. \tag{1.1}$$

Dado que todo subconjunto debe tener *algún* tamaño, de 0 a n, llegamos a la identidad

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n. \tag{1.2}$$

Tuplas y el producto cruz

Una *tupla* es una sucesión finita cuyos elementos a menudo no tienen el mismo tipo. Por ejemplo, en un plano bidimensional, un punto se puede representar con el par ordenado (x, y). Si el plano es geométrico, tanto x como y son "longitud". Pero si se trata de una gráfica de tiempo de ejecución *vs*. tamaño del problema, y podría ser segundos y x podría ser un entero. Las tuplas cortas tienen nombres especiales: par, triple, cuádruple, quíntuple, etc. En el contexto de "tupla", se sobreentiende que están ordenadas; en otros contextos, "par" podría significar "conjunto de dos" en lugar de "sucesión de dos", etc. Una k-tupla es una tupla de k elementos.

El *producto cruz* de dos conjuntos, digamos S y T, es el conjunto de pares que se pueden formar escogiendo un elemento de S como primer elemento de la tupla y un elemento de T como segundo. En notación matemática, tenemos

$$S \times T = \{(x, y) \mid x \in S, y \in T\} \tag{1.3}$$

Por tanto, $|S \times T| = |S|\,|T|$. Suele suceder que S y T son el mismo conjunto, pero esto no es necesario. Podemos definir el producto cruz iterado para generar tuplas más largas. Por ejemplo, $S \times T \times U$ es el conjunto de todas las triples que se forman tomando un elemento de S, seguido de un elemento de T, seguido de un elemento de U.

Relaciones y funciones

Una *relación* no es más que algún subconjunto de un producto cruz (posiblemente iterado). Dicho subconjunto podría ser finito o infinito, y puede estar vacío o ser todo el producto cruz. El caso más importante es una relación *binaria*, que no es sino algún subconjunto de un producto cruz simple. Todos conocemos muchos ejemplos de relaciones binarias, como "menor que" para los reales. Si $\mathbf{R}$ denota el conjunto de todos los reales, la relación "menor que" se puede definir formalmente como $\{(x, y) \mid x \in \mathbf{R}, y \in \mathbf{R}, x < y\}$. Como vemos, éste es un subconjunto de $\mathbf{R} \times \mathbf{R}$. Como ejemplo adicional, si P es el conjunto de todas las personas, entonces $P \times P$ es el conjunto de todos los pares de personas. Podemos definir "progenitor de" como (x, y) tal que x es un progenitor de y, "antepasado de" como (x, y) tal que x es un antepasado de y, y éstos son subconjuntos de $P \times P$.

Aunque muchas relaciones son pares en los que ambos elementos son del mismo tipo, la definición no lo exige. Un conjunto de pares $\{(x, y) \mid x \in S, y \in T\}$ es una relación binaria. Volviendo a nuestro ejemplo anterior de una tupla en una gráfica, una relación semejante podría representar la relación entre el tamaño del problema y el tiempo de ejecución de algún programa. Como ejemplo distinto, F podría ser el conjunto de todas las personas de sexo femenino, y entonces "x es madre de y" sería un subconjunto de $F \times P$.

Aunque las relaciones pueden ser subconjuntos arbitrarios, hay ciertas propiedades interesantes que una relación R podría tener si ambos elementos se toman del mismo conjunto subyacente, digamos S. Además, en estos casos, dado que muchas relaciones estándar tienen una notación infija (como $x < y$), es común usar la notación xRy para denotar $(x, y) \in R$.

Definición 1.2 Propiedades importantes de las relaciones

Sea $R \subseteq S \times S$. Tome nota de los significados de los términos siguientes:

reflexivo	para toda $x \in S$, $(x, x) \in R$.
simétrico	siempre que $(x, y) \in R$, (y, x) también está en R.
antisimétrico	siempre que $(x, y) \in R$, (y, x) *no* está en R.
transitivo	siempre que $(x, y) \in R$ y $(y, z) \in R$, entonces $(x, z) \in R$.

Una relación que es reflexiva, simétrica y transitiva se denomina *relación de equivalencia*, a menudo denotada con "≡". ■

Cabe señalar que "menor que" es transitiva y antisimétrica, en tanto que "menor o igual que" es transitiva y reflexiva, pero no antisimétrica (porque $x \leq x$).

Las relaciones de equivalencia son importantes en muchos problemas porque semejante relación divide el conjunto subyacente S en *particiones*; es decir, en una colección de subconjuntos disjuntos (llamados *clases de equivalencia*) S_1, S_2,..., tales que todos los elementos de S_1 son "equivalentes" entre sí, todos los elementos de S_2 son equivalentes entre sí, etc. Por ejemplo, si S es algún conjunto de enteros no negativos y definimos R como $\{(x, y) \mid x \in S, y \in S, (x - y)$ es divisible entre $3\}$, entonces R es una relación de equivalencia en S. Es evidente que $(x - x)$ es divisible entre 3. Si $(x - y)$ es divisible entre 3, también lo es $(y - x)$. Por último, si $(x - y)$ y $(y - z)$ son divisibles entre 3, también lo es $(x - z)$. Así pues, R satisface las propiedades que definen una relación de equivalencia. ¿Cómo divide R a S en particiones? Hay tres grupos, cada uno con un residuo no negativo distinto al dividir entre 3. Todos los elementos que tienen el mismo residuo son equivalentes entre sí.

Puesto que una relación binaria es un conjunto cuyos elementos son pares ordenados, a menudo conviene pensar en la relación como una tabla de dos columnas en la que cada fila contiene una tupla. Una *función* es simplemente una relación en la que ningún elemento de la primera columna se repite dentro de la relación.

Muchos problemas en los que intervienen relaciones binarias se pueden proyectar como problemas en grafos. Los problemas de grafos constituyen una clase abundante de problemas algorítmicos difíciles. Por ejemplo, en un proyecto grande que incluye muchas tareas interdependientes, podríamos tener muchos hechos de la forma "la tarea x depende de que se haya llevado a cabo la tarea y". Si un conjunto fijo de personas va a realizar las tareas, ¿cómo pueden programarse de modo que el tiempo transcurrido sea mínimo? Estudiaremos muchos problemas como éste en capítulos posteriores.

1.3.2 Herramientas de álgebra y cálculo

En esta sección presentamos algunas definiciones y propiedades elementales de logaritmos, probabilidad, permutaciones, fórmulas de sumatoria y sucesiones y series matemáticas comunes. (En este contexto, una serie es la sumatoria de una sucesión.) Introduciremos herramientas matemáticas adicionales para ecuaciones de recurrencia en el capítulo 3. El lector puede encontrar fórmulas que no se deducen aquí consultando las fuentes enumeradas en las Notas y Referencias al final del capítulo.

Funciones piso y techo

Para cualquier número real x, $\lfloor x \rfloor$ (léase "piso de x") es el entero más grande que es menor o igual que x. $\lceil x \rceil$ (léase "techo de x") es el entero más pequeño que es mayor o igual que x. Por ejemplo, $\lfloor 2.9 \rfloor = 2$ y $\lceil 6.1 \rceil = 7$.

Logaritmos

La función logaritmo, por lo regular base 2, es la herramienta matemática que más se usa en este libro. Aunque los logaritmos no son muy comunes en las ciencias naturales, son muy comunes en las ciencias de la computación.

Definición 1.3 Función logaritmo y base de logaritmo

Para $b > 1$ y $x > 0$, $\log_b x$ (léase "logaritmo base b de x") es aquel número real L tal que $b^L = x$; es decir, $\log_b x$ es la potencia a la que debemos elevar b para obtener x. ■

Las siguientes propiedades de los logaritmos se deducen fácilmente de la definición.

Lema 1.1 Sean x y y números reales positivos arbitrarios, sea a cualquier número real, y sean $b > 1$ y $c > 1$ números reales.

1. $\log_b$ es una función estrictamente creciente, es decir, si $x > y$, entonces $\log_b x > \log_b y$.
2. $\log_b$ es una función uno a uno, es decir, si $\log_b x = \log_b y$, entonces $x = y$.
3. $\log_b 1 = 0$.
4. $\log_b b^a = a$.
5. $\log_b (xy) = \log_b x + \log_b y$.
6. $\log_b (x^a) = a \log_b x$.
7. $x^{\log_b y} = y^{\log_b x}$.
8. Para convertir de una base a otra: $\log_c x = (\log_b x)/(\log_b c)$. □

Puesto que el logaritmo base 2 es el que más a menudo se usa en el campo de la complejidad computacional, existe una notación especial para denotarlo: "lg"; es decir, $\lg x = \log_2 x$. El logaritmo natural (logaritmo base e) se denota con "ln", es decir, $\ln x = \log_e x$. Cuando se usa $\log(x)$ sin indicar la base, implica que el enunciado se cumple para cualquier base.

A veces se aplica la función logaritmo a sí misma. La notación $\lg \lg(x)$ significa $\lg(\lg(x))$. La notación $\lg^{(p)}(x)$ implica p aplicaciones, de modo que $\lg^{(2)}(x)$ es lo mismo que $\lg \lg(x)$. Observe que $\lg^{(3)}(65536) = 2$, lo cual es muy diferente de $(\lg(65536))^3 = 4096$.

En casi todo el texto se obtienen logaritmos de enteros, no de números positivos arbitrarios, y a menudo necesitamos un valor entero cercano al logaritmo más que su valor exacto. Sea n un entero positivo. Si n es una potencia de 2, digamos $n = 2^k$, para algún entero k, entonces $\lg n = k$. Si n no es una potencia de 2, entonces hay un entero k tal que $2^k < n < 2^{k+1}$. En este caso, $\lfloor \lg n \rfloor = k$ y $\lceil \lg n \rceil = k + 1$. Las expresiones $\lfloor \lg n \rfloor$ y $\lceil \lg n \rceil$ se usan con frecuencia. Le recomendamos verificar estas desigualdades:

$$n \leq 2^{\lceil \lg n \rceil} < 2n.$$

$$\tfrac{n}{2} < 2^{\lfloor \lg n \rfloor} \leq 2n.$$

Por último, he aquí algunos otros hechos útiles: $\lg e \approx 1.443$ y $\lg 10 \approx 3.32$. La derivada de $\ln(x)$ es $1/x$. Utilizando la parte 8 del lema 1.1, la derivada de $\lg(x)$ es $\lg(e)/x$.

Permutaciones

Una permutación de n objetos distintos es una sucesión que contiene una vez cada uno de los objetos. Sea $S = \{s_1, s_2, \ldots, s_n\}$. Observe que los elementos de S están ordenados según sus índices; es decir, s_1 es el primer elemento, s_2 es el segundo, etc. Una permutación de S es una función uno a uno π del conjunto $\{1, 2, \ldots, n\}$ sobre sí mismo. Podemos ver π como un reacomodo de S pasando el i-ésimo elemento, s_i, a la $\pi(i)$-ésima posición. Podemos describir π con sólo enumerar sus valores, es decir, $(\pi(1), \pi(2), \ldots, \pi(n))$. Por ejemplo, para $n = 5$, $\pi = (4, 3, 1, 5, 2)$ reacomoda los elementos de S como sigue: s_3, s_5, s_2, s_1, s_4.

El número de permutaciones de n objetos distintos es $n!$. Para ver esto, observe que el primer elemento se puede pasar a cualquiera de las n posiciones; entonces esa posición queda ocupada y el segundo elemento se puede pasar a cualquiera de las $n - 1$ posiciones restantes; el tercer elemento se puede pasar a cualquiera de las $n - 2$ posiciones restantes, y así. Por tanto, el número total de posibles reacomodos es $n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1 = n!$.

Probabilidad

Supóngase que, en una situación dada, un suceso, o experimento, puede tener uno cualquiera de k desenlaces, $s_1, s_2, \ldots, s_k$. Estos desenlaces se denominan *sucesos elementales*. El conjunto de todos los sucesos elementales se llama *universo* y se denota con U. A cada desenlace s_i asociamos un número real, $Pr(s_i)$, llamado probabilidad de s_i, tal que

$$0 \leq Pr(s_i) \leq 1 \qquad \text{para } 1 \leq i \leq k;$$

$$Pr(s_1) + Pr(s_2) + \cdots + Pr(s_k) = 1.$$

Es natural interpretar $Pr(s_i)$ como el cociente del número de veces que cabe esperar que s_i ocurra y el número total de veces que se repite el experimento. (Cabe señalar, empero, que la definición no exige que las probabilidades tengan alguna correspondencia con el mundo real.) Decimos que los sucesos $s_1, \ldots, s_k$ son *mutuamente excluyentes* porque no puede ocurrir más de uno de ellos.

Los ejemplos que con mayor frecuencia se usan para ilustrar el significado de la probabilidad son el lanzamiento de monedas o dados, y diversos sucesos con naipes. De hecho, se cree que el estudio de la teoría de la probabilidad tuvo su origen en el estudio que Blaise Pascal, un matemático francés, hizo de los juegos de azar. Si el "experimento" es el lanzamiento de una moneda,

ésta puede caer con la "cara" hacia arriba o con el "sello" hacia arriba. Sea s_1 = "cara" y s_2 = "sello", y asignemos $Pr(s_1) = 1/2$ y $Pr(s_2) = 1/2$. (Si alguien objeta porque la moneda podría caer de canto, podríamos definir s_3 = "canto" y asignar $Pr(s_3) = 0$. Sin embargo, con un número finito de sucesos, podemos hacer caso omiso de un suceso con probabilidad cero, por lo que normalmente no se definen tales sucesos elementales.) Si se lanza un dado de seis caras, hay seis posibles desenlaces: para $1 \leq i \leq 6$, s_i = "el dado cae con la cara número i hacia arriba", y $Pr(s_i) = 1/6$. En general, si hay k posibles desenlaces y todos se consideran igualmente verosímiles, asignamos $Pr(s_i) = 1/k$ para cada i. Con frecuencia no hay razón para suponer que todos los desenlaces tienen la misma probabilidad; tal supuesto suele usarse en ejemplos o en casos en los que no hay datos que apoyen un supuesto mejor.

Si en el experimento intervienen varios objetos, un suceso elemental deberá tomar en cuenta lo que se observa acerca de todos ellos. Por ejemplo, si se lanzan dos dados, A y B, el suceso "A cae con el lado 1 hacia arriba" no es un suceso elemental porque hay varios desenlaces asociados a B. En este caso, los sucesos elementales serían s_{ij} = "el dado A cae con el lado i hacia arriba y el dado B cae con el lado j hacia arriba", para $1 \leq i, j \leq 6$. Abreviaremos esta descripción a "A cae i y B cae j" de aquí en adelante. Hay 36 sucesos elementales, y se acostumbra asignar una probabilidad de $1/36$ a cada uno.

A menudo es necesario considerar la probabilidad de que ocurra cualquiera de varios desenlaces especificados o de que el desenlace tenga una propiedad dada. Sea S un subconjunto de los sucesos elementales $\{s_1, \ldots, s_k\}$. Entonces decimos que S es un *suceso*, y $Pr(S) = \sum_{s_i \in S} Pr(s_i)$. Por ejemplo, supóngase que se lanza un dado, y definimos el suceso S como "el número que sale es divisible entre 3". Entonces, la probabilidad de S es $Pr(S) = Pr(\{s_3, s_6\}) = Pr(s_3) + Pr(s_6) = 1/3$. Los sucesos elementales también son sucesos.

Dos sucesos especiales son el *suceso seguro*, $U = \{s_1, \ldots, s_k\}$, cuya probabilidad es 1, y el *suceso imposible*, $\varnothing$, cuya probabilidad es 0. (Recuerde que $\varnothing$ denota el conjunto vacío.) También, para cualquier suceso S, existe el suceso complemento "no S", que consiste en todos los sucesos elementales que no están en S, es decir, $U - S$. Desde luego, $Pr(\text{no } S) = 1 - Pr(S)$.

Los sucesos se pueden definir en términos de otros sucesos utilizando los conectores lógicos "y" y "o". El suceso "S_1 y S_2" es $(S_1 \cap S_2)$, la intersección de S_1 y S_2. El suceso "S_1 o S_2" es $(S_1 \cup S_2)$, la unión de S_1 y S_2.

A menudo necesitamos analizar probabilidades basadas en cierto grado de conocimiento parcial acerca del experimento. Éstas se denominan *probabilidades condicionales*.

Definición 1.4 Probabilidad condicional

La *probabilidad condicional de un suceso S dado un suceso T* se define como

$$Pr(S \mid T) = \frac{Pr(S \text{ y } T)}{Pr(T)} = \frac{\sum_{s_i \in S \cap T} Pr(s_i)}{\sum_{s_j \in T} Pr(s_j)}, \tag{1.4}$$

donde s_i y s_j cubren intervalos de sucesos elementales. ■

Ejemplo 1.3 Probabilidad condicional con dos dados

Supóngase que en el experimento se lanzan dos dados, A y B. Definamos tres sucesos:

S_1: "*A* cae 1",

S_2: "*B* cae 6",

S_3: "La suma de los números que salen es 4 o menos".

Para tener una idea intuitiva del significado de la probabilidad condicional, consideremos el caso sencillo en el que todos los sucesos elementales tienen la misma probabilidad. En nuestro ejemplo, los 36 sucesos elementales son de la forma "*A* cae *i* y *B* cae *j*", para $1 \leq i, j \leq 6$. Entonces la probabilidad condicional $Pr(S_1 \mid S_3)$ se puede interpretar como la respuesta a la pregunta, "De todos los sucesos elementales de S_3, ¿qué fracción de esos sucesos elementales está también en S_1?"

Enumeremos todos los sucesos elementales de S_3:

"*A* cae 1 y *B* cae 1", "*A* cae 2 y *B* cae 1",

"*A* cae 1 y *B* cae 2", "*A* cae 2 y *B* cae 2",

"*A* cae 1 y *B* cae 3", "*A* cae 3 y *B* cae 1".

El suceso S_1 consiste en 6 sucesos elementales en los que *A* cae 1 y *B* cae cada uno de sus seis posibles valores. Tres de los sucesos elementales de S_3 están también en S_1, así que la respuesta a la pregunta es $3/6 = 1/2$. Mediante un cálculo exacto con la fórmula de la ecuación (1.4), la probabilidad de S_1 dado S_3 es

$$Pr(S_1 \mid S_3) = \frac{3/36}{6/36} = 1/2.$$

Observe que la probabilidad condicional de S_2 dado S_3 es 0; es decir, $Pr(S_2 \mid S_3) = 0$. ■

En general, el procedimiento para calcular probabilidades condicionales dado algún suceso específico *S* consiste en eliminar todos los sucesos elementales que no están en *S* y luego ajustar la escala de las probabilidades de todos los sucesos elementales restantes utilizando el mismo factor, de modo que la sumatoria de las nuevas probabilidades sea 1. El factor requerido es $1/Pr(S)$.

La probabilidad condicional de un suceso puede ser mayor o menor que la probabilidad incondicional de ese suceso. En el ejemplo 1.3 la probabilidad incondicional de S_1 es $1/6$ y la probabilidad condicional de S_1 dado S_3 es $1/2$. Por otra parte, la probabilidad incondicional de que "el número en que *A* cae sea divisible entre 3" es de $1/3$; pero en el ejemplo 1.3 vemos que la probabilidad condicional de que "el número en que *A* cae sea divisible entre 3" dado S_3 es $1/6$.

Definición 1.5 Independencia estocástica

Dados dos sucesos *S* y *T*, si

$$Pr(S \text{ y } T) = Pr(S)Pr(T)$$

entonces *S* y *T* son *estocásticamente independientes*, o simplemente *independientes*. ■

Si *S* es estocásticamente independiente de *T*, entonces $Pr(S \mid T) = Pr(S)$ (véase el ejercicio 1.8). Es decir, saber que ha ocurrido el suceso *T* no influye en la probabilidad de que ocurra el su-

ceso S, en un sentido o en otro. La propiedad de independencia es extremadamente útil si existe, pues permite analizar por separado las probabilidades de que ocurran sucesos diferentes. Por otra parte, se cometen muchas equivocaciones en los análisis cuando el supuesto de independencia es injustificado.

Ejemplo 1.4 Independencia estocástica

Continuando con los sucesos definidos en el ejemplo 1.3, los sucesos S_1 y S_2 son independientes porque la probabilidad de cada uno es de $1/6$, y (S_1 y S_2) consiste en un suceso elemental, cuya probabilidad es $1/36$. Observe también que $Pr(S_1 \mid S_2) = (1/36)/(6/36) = 1/6 = Pr(S_1)$.

De la explicación del ejemplo 1.3, vemos que S_1 y S_3 no son independientes, y que S_2 y S_3 no son independientes. ■

Las variables aleatorias y sus valores esperados son importantes en muchas situaciones que implican probabilidades. Una variable aleatoria es una variable con valor real que depende de qué suceso elemental ha ocurrido; dicho de otro modo, es una función definida para sucesos elementales. Por ejemplo, si el número de operaciones efectuadas por un algoritmo depende de las entradas, y cada posible entrada es un suceso elemental, entonces el número de operaciones será una variable aleatoria.

Definición 1.6 Expectativa y expectativa condicional

Sea $f(e)$ una variable aleatoria definida sobre un conjunto de sucesos elementales $e \in U$. La *expectativa* de f, denotada por $E(f)$, se define como

$$E(f) = \sum_{e \in U} f(e)Pr(e).$$

Esto también se conoce como *valor promedio* de f. La *expectativa condicional* de f dado un suceso S, denotada por $E(f \mid S)$, se define como

$$E(f \mid S) = \sum_{e \in U} f(e)Pr(e \mid S) = \sum_{e \in S} f(e)Pr(e \mid S)$$

puesto que la probabilidad condicional de cualquier suceso que no está en S es 0. ■

Las expectativas suelen ser más fáciles de manipular que las variables aleatorias mismas, sobre todo cuando intervienen múltiples variables aleatorias interrelacionadas, debido a ciertas leyes importantes que presentamos a continuación y que se demuestran fácilmente a partir de las definiciones.

Lema 1.2 (Leyes de expectativas) Para las variables aleatorias $f(e)$ y $g(e)$ definidas sobre un conjunto de sucesos elementales $e \in U$, y cualquier suceso S:

$$E(f + g) = E(f) + E(g),$$
$$E(f) = Pr(S)E(f \mid S) + Pr(\text{no } S)\, E(f \mid \text{no } S). \quad \square$$

Ejemplo 1.5 Probabilidad condicional y orden

En el capítulo 4 consideraremos probabilidades en relación con información de orden obtenida efectuando comparaciones. Veamos un ejemplo de ese tipo en el que intervienen cuatro elemen-

tos A, B, C, D, que tienen valores numéricos distintos, aunque inicialmente no sabemos nada acerca de sus valores ni de sus valores relativos. Escribiremos las letras en orden para denotar el suceso elemental de que ése sea su orden relativo; es decir, *CBDA* es el suceso de que $C < B < D < A$. Hay 24 posibles permutaciones:

ABCD	*ACBD*	*CABD*	*ACDB*	*CADB*	*CDAB*
ABDC	*ADBC*	*DABC*	*ADCB*	*DACB*	*DCAB*
BACD	*BCAD*	*CBAD*	*BCDA*	*CBDA*	*CDBA*
BADC	*BDAC*	*DBAC*	*BDCA*	*DBCA*	*DCBA*

Comenzamos por suponer que todas las permutaciones introducidas son igualmente probables, así que la probabilidad de cada una es $1/24$. ¿Cuál es la probabilidad de que $A < B$? Dicho de otro modo, definiendo $A < B$ como un suceso, ¿cuál es su probabilidad? Intuitivamente esperamos que la probabilidad sea $1/2$, y podemos verificarlo contando el número de permutaciones en las que A aparece antes de B en la sucesión. De forma similar, para cualquier par de elementos, la probabilidad de que uno sea menor que otro es de $1/2$. Por ejemplo, el suceso $B < D$ tiene probabilidad $1/2$.

Supóngase ahora que el programa *compara* A y B y descubre que $A < B$. ¿Cómo "afecta" esto las probabilidades? Para hacer más rigurosa esta pregunta, la plantearemos así: "¿Cuáles son las probabilidades condicionadas al suceso $A < B$?" Por inspección, vemos que el suceso $A < B$ consiste en todos los sucesos elementales de las dos primeras filas de la tabla. Por tanto, las probabilidades condicionales de estos sucesos elementales dado $A < B$ son el doble de sus probabilidades originales, $2/24 = 1/12$, mientras que las probabilidades condicionales de los sucesos elementales dado $A < B$ (las dos filas inferiores) son 0.

Recuerde que antes de efectuar comparaciones, la probabilidad del suceso $B < D$ era $1/2$. No hemos comparado B y D. ¿Sigue siendo $1/2$ la probabilidad condicional de $B < D$, dado $A < B$? Para contestar la pregunta, verificamos en cuántas sucesiones de las dos primeras filas B está antes de D. De hecho, sólo hay cuatro casos en los que B precede a D en las dos primeras filas, así que $Pr(B < D \mid A < B) = 1/3$.

Consideremos ahora el suceso $C < D$. ¿Su probabilidad condicional es diferente de $1/2$? Revisando una vez más las dos primeras filas de la tabla, vemos que C precede a D en seis casos, así que $Pr(C < D \mid A < B) = 1/2$. Por tanto, los sucesos $A < B$ y $C < D$ son estocásticamente independientes. Esto es lo que cabría esperar: el orden relativo de A y B no deberá "influir" en el orden de C y D.

Por último, supóngase que el programa efectúa otra comparación y descubre que $D < C$ (ya descubrió que $A < B$). Examinemos las probabilidades condicionales dados ambos sucesos (lo que también es el suceso individual "$A < B$ y $D < C$"). Vemos por inspección que el suceso "$A < B$ y $D < C$" consiste en todos los sucesos elementales de la segunda fila de la tabla. Para hacer que la sumatoria de las probabilidades condicionales sea 1, todos estos sucesos elementales deben tener una probabilidad condicional de $1/6$. El programa no ha comparado A o B ni con C ni con D. ¿Implica esto que las probabilidades condicionales de los sucesos $A < C$, $A < D$, $B < C$ y $B < D$ no han cambiado respecto a sus probabilidades originales, todas las cuales eran $1/2$? La respuesta se deduce en el ejercicio 1.10. ■

Ejemplo 1.6 Número esperado de inversiones

Consideremos el mismo espacio de probabilidades del ejemplo 1.5. Definamos la variable aleatoria $I(e)$ como el número de pares de elementos cuyo orden de claves relativo es opuesto a su orden alfabético. Esto es el número de *inversiones* en la permutación. Por ejemplo, $I(ABCD) = 0$, $I(ABDC) = 1$ porque $D < C$ pero C está antes que D en el orden alfabético, $I(DCBA) = 6$, etc. Por inspección, vemos que $E(I) = 3$. Consideremos ahora $E(I \mid A < B)$ y $E(I \mid B < A)$. Una vez más, por conteo directo vemos que son 2.5 y 3.5, respectivamente. Puesto que $Pr(A < B) = Pr(B < A) = \frac{1}{2}$, el lema 1.2 nos dice que $E(I) = \frac{1}{2}(2.5 + 3.5)$, lo cual es cierto. ■

En síntesis, las probabilidades condicionales reflejan las incertidumbres de una situación cuando tenemos un conocimiento parcial. Se pueden calcular desechando todos los sucesos elementales que sabemos son imposibles en la situación actual, y ajustando después la escala de las probabilidades de los sucesos elementales restantes de modo que su sumatoria vuelva a ser 1. Cualquier suceso cuya probabilidad *no* cambie como resultado de este cálculo es (estocásticamente) independiente del suceso conocido. Los sucesos independientes a menudo implican objetos que no tienen influencia unos sobre otros (como múltiples monedas o múltiples dados).

Sumatorias y series

Hay varias sumatorias que se presentan con frecuencia en el análisis de algoritmos. Presentaremos aquí y en la sección que sigue las fórmulas de algunas de ellas, con breves sugerencias que podrían ayudar al lector a recordarlas. Una nota acerca de la terminología: una *serie* es la sumatoria de una *sucesión*.

Series aritméticas: La sumatoria de enteros consecutivos:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{1.5}$$

Cómo recordarla: Escriba los enteros del 1 a n. Aparee el primero y el último, es decir, 1 y n; aparee el segundo y el penúltimo, 2 y $n - 1$, y así. Cada par sumado da $(n + 1)$ y hay $n/2$ pares, lo que da el resultado. (Si n es impar, el elemento central cuenta como "medio par".) El mismo truco funciona con límites distintos de 1 y n.

Series polinómicas: Primero, consideramos la suma de cuadrados.

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6}. \tag{1.6}$$

Esto se puede probar por inducción sobre n. Lo principal que debemos recordar es que la suma de los primeros n cuadrados es aproximadamente $n^3/3$. No usaremos la ecuación (1.6) en el texto, pero se podría necesitar en algunos de los ejercicios.

El caso general es

$$\sum_{i=1}^{n} i^k \approx \frac{1}{k+1} n^{k+1}, \tag{1.7}$$

que se justifica por aproximación con una integral, como se describe en la sección que sigue. (Para cualquier k específica, se puede demostrar una fórmula exacta por inducción.) Comparemos este tipo de serie cuidadosamente con las "series geométricas", a continuación.

Potencias de 2: Éste es un caso común de *serie geométrica*.

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1. \tag{1.8}$$

Cómo recordarla: Piense en cada término 2^i como un bit 1 en un número binario; entonces:

$$\sum_{i=0}^{k} 2^i = 11\ldots1.$$

Hay $k + 1$ bits 1. Si sumamos 1 a este número, el resultado es

$$100\ldots0 = 2^{k+1}.$$

(Este resultado también puede obtenerse utilizando la fórmula siguiente para las series geométricas.)

Series geométricas:

$$\sum_{i=0}^{k} a\, r^i = a\left(\frac{r^{k+1} - 1}{r - 1}\right). \tag{1.9}$$

Para verificar esto, elimine por división el miembro derecho. Como caso especial, con $r = \frac{1}{2}$, tenemos

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k}. \tag{1.10}$$

Una serie geométrica se distingue por tener una constante en la base y una variable en el exponente. Una serie polinómica tiene una variable en la base y un exponente constante. Los comportamientos son muy distintos.

Series armónicas:

$$\sum_{i=1}^{k} \frac{1}{i} \approx \ln(n) + \gamma, \qquad \text{donde } \gamma = 0.577. \tag{1.11}$$

La sumatoria se denomina n-ésimo número Armónico. La constante γ se llama *constante de Euler*. Véase también el ejemplo 1.7.

Series aritmético-geométricas: En la sumatoria siguiente, el término i nos daría una serie aritmética y el término 2^i nos daría una serie geométrica, de ahí el nombre.

$$\sum_{i=1}^{k} i\, 2^i = (k - 1)2^{k+1} + 2. \tag{1.12}$$

La deducción es un ejemplo de "sumatoria por partes", que es análoga a la "integración por partes". La sumatoria se reacomoda para dar una diferencia de dos sumatorias que se cancelan con excepción de sus términos primero y último, menos una tercera sumatoria de una forma más sencilla:

$$\begin{aligned}\sum_{i=1}^{k} i\,2^i &= \sum_{i=1}^{k} i(2^{i+1} - 2^i)\\ &= \sum_{i=1}^{k} i\,2^{i+1} - \sum_{i=0}^{k-1} (i+1)2^{i+1}\\ &= \sum_{i=1}^{k} i\,2^{i+1} - \sum_{i=0}^{k-1} i\,2^{i+1} - \sum_{i=0}^{k-1} 2^{i+1}\\ &= k\,2^{k+1} - 0 - (2^{k+1} - 2) = (k-1)2^{k+1} + 2.\end{aligned}$$

Números de Fibonacci: La sucesión de Fibonacci se define recursivamente como:

$$\begin{aligned}&F_n = F_{n-1} + F_{n-2} \qquad \text{para } n \geq 2,\\ &F_0 = 0, \quad F_1 = 1.\end{aligned} \tag{1.13}$$

Aunque no se trata de una sumatoria, la serie se presenta con frecuencia en el análisis de algoritmos.

Funciones monotónicas y convexas

En ocasiones, bastan propiedades muy generales para sacar algunas conclusiones útiles acerca del comportamiento de las funciones. Dos de esas propiedades son la *monotonicidad* y la *convexidad*. En toda la explicación de la monotonicidad y la convexidad en esta sección, se sobreentiende algún intervalo $a \leq x < \infty$, donde a suele ser 0 pero podría ser 1 si se trata de logaritmos. Todos los puntos mencionados están en este intervalo, y f está definida en dicho intervalo. El dominio podría ser los reales o los enteros.

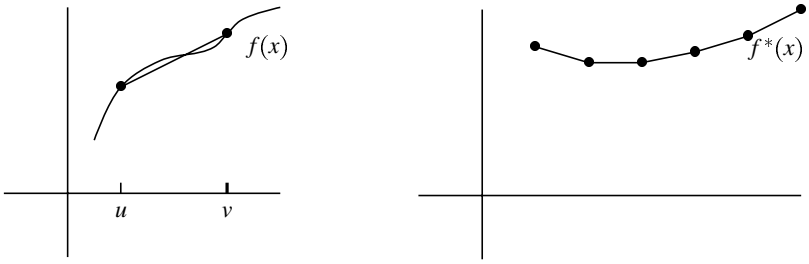
Definición 1.7 Funciones monotónicas y antimonotónicas

Decimos que una función $f(x)$ es *monotónica*, o *no decreciente*, si $x \leq y$ siempre implica que $f(x) \leq f(y)$. Una función $f(x)$ es *antimonotónica*, o *no creciente*, si $-f(x)$ es monotónica. ■

Los siguientes son ejemplos de funciones monotónicas familiares: x, x^2 para $x \geq 0$, $\log(x)$ para $x > 0$ y e^x. Las siguientes son funciones monotónicas menos familiares: $\lfloor x \rfloor$ y $\lceil x \rceil$, lo que demuestra que las funciones monotónicas no tienen que ser continuas. Un ejemplo antimonotónico es $1/x$ para $x > 0$.

Definición 1.8 Función de interpolación lineal

La *interpolación lineal* de una función dada $f(x)$ entre dos puntos u y v, $u < v$, es la función definida por



(a) Interpolación lineal (b) Extensión de $f(n)$ a $f^*(x)$

Figura 1.3 Ilustraciones para la explicación de convexidad: la función f es diferente en las partes (a) y (b). En la parte (b), $f^*(x)$ es convexa.

$$L_{f,u,v}(x) = \frac{(v - x)\, f(u) + (x - u)\, f(v)}{(v - u)}$$

$$= f(u) + (x - u)\, \frac{f(v) - f(u)}{v - u} = f(v) - (v - x)\, \frac{f(v) - f(u)}{v - u}, \qquad (1.14)$$

es decir, el segmento de línea recta que une $f(u)$ y $f(v)$ (véase la figura 1.3a). ■

Definición 1.9 Funciones convexas

Decimos que una función $f(x)$ es *convexa* si para todo $u < v$, $f(x) \leq L_{f,u,v}(x)$ en el intervalo (u, v). Informalmente, $f(x)$ es convexa si nunca se curvea hacia abajo. ■

Así pues, las funciones como x, x^2, $1/x$ y e^x son convexas. La función de la figura 1.3(b) es convexa (pero no monotónica), sea que se interprete en los reales o únicamente en los enteros; la función de la figura 1.3(a) es monotónica, pero no convexa. Tampoco $\log(x)$ ni $\sqrt{x}$ son convexas. ¿Y $x \log(x)$? Los lemas siguientes desarrollan algunas pruebas de convexidad prácticas. Es fácil ver (y es posible demostrar) que una función discontinua no puede ser convexa. El lema 1.3 dice que basta considerar puntos uniformemente espaciados para probar si una función es convexa o no, lo que simplifica considerablemente la tarea. La demostración es el ejercicio 1.16.

Lema 1.3

1. Sea $f(x)$ una función continua definida en los reales. Entonces $f(x)$ es convexa si y sólo si, para cualesquier puntos x, y,

$$f(\tfrac{1}{2}(x + y)) \leq \tfrac{1}{2}\,(f(x) + f(y)).$$

Dicho con palabras, f evaluada en el punto medio entre x y y está en el punto medio de la interpolación lineal de f entre x y y o debajo de ese punto. Observe que el punto medio de la interpolación lineal no es más que el promedio de $f(x)$ y $f(y)$.

2. Una función $f(n)$ definida en los enteros es convexa si y sólo si, para cualquier n, $n + 1$, $n + 2$,

$$f(n + 1) \leq \frac{1}{2}(f(n) + f(n + 2)).$$

Dicho con palabras, $f(n + 1)$ es cuando más el promedio de $f(n)$ y $f(n + 2)$. □

El lema 1.4 resume varias propiedades útiles de la monotonicidad y la convexidad: dice que las funciones definidas sólo en los enteros se pueden extender a los reales por interpolación lineal, conservando las propiedades de monotonicidad y convexidad. También se plantean algunas propiedades que tienen que ver con derivadas. Las demostraciones están en los ejercicios 1.17 a 1.19.

Lema 1.4

1. Sea $f(n)$, definida únicamente en los enteros. Sea $f*(x)$ la extensión de f a los reales por interpolación lineal entre enteros consecutivos (véase la figura 1.3b).
   a. $f(n)$ es monotónica si y sólo si $f*(x)$ es monotónica.
   b. $f(n)$ es convexa si y sólo si $f*(x)$ es convexa.
2. Si la primera derivada de $f(x)$ existe y no es negativa, entonces $f(x)$ es monotónica.
3. Si la primera derivada de $f(x)$ existe y es monotónica, entonces $f(x)$ es convexa.
4. Si la segunda derivada de $f(x)$ existe y no es negativa, entonces $f(x)$ es convexa. (Esto se sigue de las partes 2 y 3.) □

Sumatorias empleando integración

Varias sumatorias que surgen con frecuencia en el análisis de algoritmos se pueden aproximar (o acotar desde arriba o desde abajo) empleando integración. Primero repasemos algunas fórmulas de integración útiles:

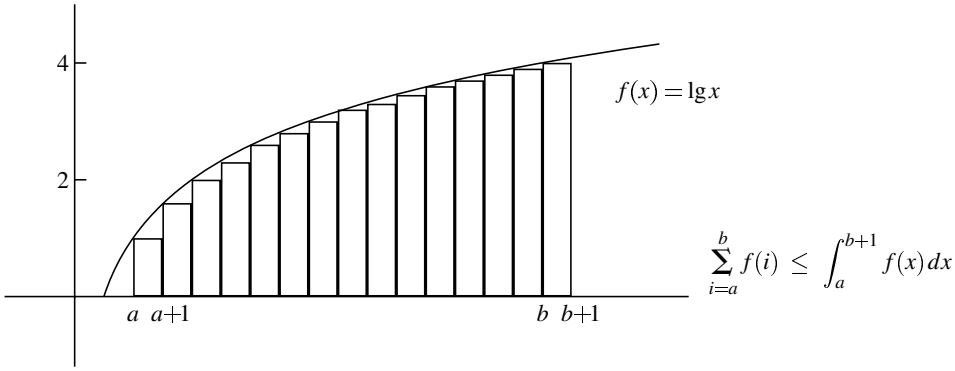
$$\int_0^n x^k\,dx = \frac{1}{k+1}n^{k+1}. \qquad \int_0^n e^{ax}\,dx = \frac{1}{a}(e^{an} - 1).$$
$$\int_1^n x^k \ln(x)\,dx = \frac{1}{k+1}n^{k+1}\ln(n) - \frac{1}{(k+1)^2}n^{k+1}. \tag{1.15}$$

Si $f(x)$ es monotónica (o no decreciente), entonces

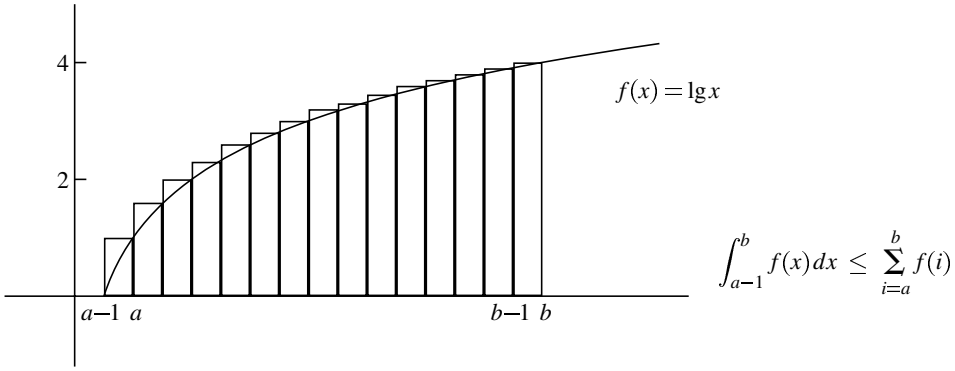
$$\int_{a-1}^{b} f(x)\,dx \leq \sum_{i=a}^{b} f(i) \leq \int_a^{b+1} f(x)dx. \tag{1.16}$$

De forma similar, si $f(x)$ es antimonotónica (o no creciente), entonces

$$\int_a^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)\,dx. \tag{1.17}$$



(a) Sobreaproximación



(b) Subaproximación

Figura 1.4 Aproximación de una sumatoria de valores de una función monotónica (o no decreciente)

Esta situación para $f(x)$ monotónica se ilustra en la figura 1.4. He aquí dos ejemplos que usaremos posteriormente en el texto.

Ejemplo 1.7 Un estimado de $\sum_{i=1}^{n} \frac{1}{i}$

$$\sum_{i=1}^{n} \frac{1}{i} \leq 1 + \int_{1}^{n} \frac{dx}{x} = 1 + \ln x \,|_{1}^{n} = 1 + \ln n - \ln 1 = \ln(n) + 1.$$

usando la ecuación (1.17). Observe que separamos el primer término de la sumatoria y aplicamos la aproximación de integral al resto, para evitar una división entre cero en el límite de integración inferior. De forma similar,

$$\sum_{i=1}^{n} \frac{1}{i} \geq \ln(n+1).$$

La ecuación (1.11) da una aproximación más exacta. ■

Ejemplo 1.8 Una cota inferior para $\sum_{i=1}^{n} \lg i$

$$\sum_{i=1}^{n} \lg i = 0 + \sum_{i=2}^{n} \lg i \geq \int_{1}^{n} \lg x \, dx$$

por la ecuación (1.16) (véase la figura 1.4b). Ahora bien,

$$\begin{aligned}\int_{1}^{n} \lg x \, dx &= \int_{1}^{n} (\lg e) \ln x \, dx = (\lg e) \int_{1}^{n} \ln x \, dx \\ &= (lg\ e)(x \ln x - x)\Big|_{1}^{n} = (\lg e)(n \ln n - n + 1) \\ &= n \lg n - n \lg e + \lg e \geq n \lg n - n \lg e.\end{aligned}$$

Puesto que $\lg e < 1.443$,

$$\sum_{i=1}^{n} \lg i \geq n \lg n - 1.443n \tag{1.18}$$

■

Utilizando las ideas del ejemplo anterior, pero con matemáticas más precisas, es posible deducir la *fórmula de Stirling* que da cotas para $n!$:

$$\left(\frac{n}{e}\right)^n \sqrt{2\pi n} < n! < \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \frac{1}{11n}\right) \qquad \text{para } n \geq 1. \tag{1.19}$$

Manipulación de desigualdades

Las reglas que siguen para combinar desigualdades son a menudo útiles:

Transitividad		*Suma*		*Amplificación*		
Si	$A \leq B$	Si	$A \leq B$	Si	$A \leq B$	
y	$B \leq C$	y	$C \leq D$	y	$\alpha > 0$	(1.20)
entonces	$A \leq C$	entonces	$A + C \leq B + D$	entonces	$\alpha A \leq \alpha B$	

1.3.3 Elementos de lógica

La lógica es un sistema para formalizar enunciados en lenguaje natural y así razonar con mayor exactitud. Los enunciados más sencillos se denominan *fórmulas atómicas*. Es posible formar enunciados más complejos empleando *conectores lógicos*. Los siguientes son ejemplos de fórmulas atómicas: "$4 > 3$", "4.2 es un entero" y "$x + 1 > x$". Cabe señalar que un enunciado lógico no tiene que ser verdad. El objetivo de una demostración es mostrar que un enunciado lógico es verdad.

Los conectores lógicos más conocidos son "$\wedge$" (y), "$\vee$" (o) y "$\neg$" (no), que también se denominan operadores booleanos. El valor de verdad de un enunciado complejo se deduce de los valores de verdad de sus fórmulas atómicas, según las reglas de los conectores. Sean A y B enunciados lógicos. Entonces,

1. $A \wedge B$ es verdadero si y sólo si A es verdadero y B es verdadero;
2. $A \vee B$ es verdadero si y sólo si A es verdadero o B es verdadero, o ambos lo son;
3. $\neg A$ es verdadero si y sólo si A es falso.

Otro conector importante para el razonamiento se denomina "implica", que denotamos con el símbolo "$\Rightarrow$". (También se usa el símbolo "$\rightarrow$".) El enunciado $A \Rightarrow B$ se lee "A implica B", o "si A, entonces B". (Cabe señalar que este enunciado no tiene cláusula "si no".) El operador "implica" se puede representar con una combinación de otros operadores, según la identidad:

$$A \Rightarrow B \quad \text{equivale lógicamente a} \quad \neg A \vee B. \tag{1.21}$$

Esto se puede verificar revisando todas las combinaciones de asignaciones de verdad a A y B.

Otro conjunto de identidades útil recibe el nombre de *leyes de DeMorgan*:

$$\neg(A \wedge B) \quad \text{equivale lógicamente a} \quad \neg A \vee \neg B, \tag{1.22}$$

$$\neg(A \vee B) \quad \text{equivale lógicamente a} \quad \neg A \wedge \neg B. \tag{1.23}$$

Cuantificadores

Otro tipo importante de conector lógico es el *cuantificador*. El símbolo $\forall x$ se denomina *cuantificador universal* y se lee "para toda x", mientras que el símbolo $\exists x$ es el *cuantificador existencial* y se lee "existe x". Estos conectores se pueden aplicar a enunciados que contienen la variable x. El enunciado $\forall x P(x)$ es verdadero si y sólo si $P(x)$ es verdadero para toda x. El enunciado $\exists x P(x)$ es verdadero si y sólo si $P(x)$ es verdadero para *algún* valor de x. Lo más común es que un enunciado cuantificado universalmente sea condicional: $\forall x(A(x) \Rightarrow B(x))$. Esto se puede leer así: "para toda x tal que, si $A(x)$ se cumple, $B(x)$ se cumple".

Los enunciados cuantificados obedecen una variación de las leyes de DeMorgan:

$$\forall x\, A(x) \quad \text{equivale lógicamente a} \quad \neg\exists x\, (\neg A(x)), \tag{1.24}$$

$$\exists x\, A(x) \quad \text{equivale lógicamente a} \quad \neg\forall x\, (\neg A(x)). \tag{1.25}$$

Hay ocasiones en que no es fácil efectuar la traducción del lenguaje natural a un enunciado cuantificado. Las personas no suelen hablar en el artificioso lenguaje de la lógica. Debemos tener en cuenta que "para cualquier x" normalmente significa "para toda x", aunque "cualquier" y "alguna" a menudo se pueden usar de manera intercambiable en el habla común y corriente. La me-

jor pauta es tratar de replantear un enunciado en lenguaje natural de modo que se acerque más a la forma lógica, y luego preguntarse si significa lo mismo en el lenguaje natural. Por ejemplo, si el enunciado original es "Cualquier persona debe respirar para vivir". Esto podría replantearse como "Para todas las personas x, x debe respirar para vivir" o como "Para alguna persona x, x debe respirar para vivir". ¿Cuál significa lo mismo que el enunciado original?

Negación de un enunciado cuantificado, contraejemplos

¿Qué se necesita para demostrar que un enunciado general, digamos $\forall x(A(x) \Rightarrow B(x))$, es falso? Podemos usar las identidades anteriores para aclarar el objetivo. La primera cosa de la que hay que darse cuenta es que *no* es necesario demostrar $\forall x(A(x) \Rightarrow \neg B(x))$. Tal enunciado sería demasiado fuerte. La negación de $\forall x(A(x) \Rightarrow B(x))$ es $\neg(\forall x(A(x) \Rightarrow B(x)))$, que se puede someter a una serie de transformaciones:

$$\begin{aligned} \neg(\forall x(A(x) \Rightarrow B(x))) \quad & \text{equivale lógicamente a} \quad \exists x \neg(A(x) \Rightarrow B(x)) \\ & \text{equivale lógicamente a} \quad \exists x \neg(\neg A(x) \vee B(x)) \\ & \text{equivale lógicamente a} \quad \exists x(A(x) \wedge \neg B(x)). \end{aligned} \tag{1.26}$$

Dicho en palabras, si podemos mostrar *algún* objeto x para el cual $A(x)$ sea verdadero y $B(x)$ sea falso, habremos demostrado que $\forall x(A(x) \Rightarrow B(x))$ es falso. Semejante objeto (x) es un *contraejemplo*.

Contrapositivos

Al tratar de demostrar un enunciado, a menudo es conveniente manipularlo para convertirlo en una forma lógicamente equivalente. Una de esas formas es el *contrapositivo*. El *contrapositivo* de $A \Rightarrow B$ es $(\neg B) \Rightarrow (\neg A)$. La ecuación (1.21) nos permite verificar que el *contrapositivo* de una implicación es verdadero exactamente cuando la implicación en sí es verdadera:

$$A \Rightarrow B \quad \text{equivale lógicamente a} \quad (\neg B) \Rightarrow (\neg A). \tag{1.27}$$

Hay quienes llaman "demostración por contradicción" al hecho de demostrar el contrapositivo de un enunciado, pero "demostración por contraposición" sería una descripción más exacta. A continuación describimos la "demostración por contradicción" genuina.

Demostración por contradicción

Supóngase que lo que se busca es demostrar un enunciado de la forma $A \Rightarrow B$. Una *demostración por contradicción* genuina agrega una hipótesis adicional de $\neg B$, y luego demuestra B mismo. Es decir, $(A \wedge \neg B) \Rightarrow B$ es el enunciado completo que se demuestra. La identidad siguiente justifica este método:

$$A \Rightarrow B \quad \text{equivale lógicamente a} \quad (A \wedge \neg B) \Rightarrow B. \tag{1.28}$$

Las demostraciones por contradicción genuinas son muy poco comunes en el análisis de algoritmos. No obstante, el ejercicio 1.21 requiere una. La mayor parte de las llamadas demostraciones por contradicción son en realidad demostraciones por contraposición.

Reglas de inferencia

Hasta ahora hemos visto muchos pares de enunciados *lógicamente equivalentes*, o *identidades lógicas*: un enunciado es verdadero si y sólo si el segundo enunciado es verdadero. Las identidades

son "reversibles". Sin embargo, la mayor parte de las demostraciones se refieren a combinaciones "irreversibles" de enunciados. El enunciado completo a demostrar tiene la forma "si *hipótesis*, entonces *conclusión*". La reversión, "si conclusión, entonces hipótesis" a menudo *no* se cumple. Las identidades lógicas no son lo bastante flexibles como para demostrar semejantes enunciados "si-entonces". En tales situaciones, se necesitan *reglas de inferencia*.

Una regla de inferencia es un patrón general que nos permite sacar alguna conclusión *nueva* de un conjunto de enunciados dado. Se puede decir, "Si sabemos que $B_1, \ldots, B_k$, entonces podemos concluir C", donde $B_1, \ldots, B_k$ y C son enunciados lógicos por derecho propio. He aquí unas cuantas reglas muy conocidas:

Si sabemos que			*entonces podemos concluir que*	
B	y	$B \Rightarrow C$	C	(1.29)
$A \Rightarrow B$	y	$B \Rightarrow C$	$A \Rightarrow C$	(1.30)
$B \Rightarrow C$	y	$\neg B \Rightarrow C$	C	(1.31)

Algunas de estas reglas se conocen por su nombre en griego o en latín. La ecuación (1.29) es *modus ponens*, la ecuación (1.30) es *silogismo* y la ecuación (1.31) es la *regla de casos*. Estas reglas no son independientes; en el ejercicio 1.21 el lector demostrará la regla de casos utilizando otras reglas de inferencia e identidades lógicas.

1.4 Análisis de algoritmos y problemas

Analizamos los algoritmos con la intención de mejorarlos, si es posible, y de escoger uno de entre varios con los que se podría resolver un problema. Usaremos los criterios siguientes:

1. Corrección
2. Cantidad de trabajo realizado
3. Cantidad de espacio usado
4. Sencillez, claridad
5. Optimidad

Analizaremos cada uno de estos criterios a fondo y daremos varios ejemplos de su aplicación. Al considerar la optimidad de los algoritmos, presentaremos técnicas para establecer una cota o límite inferior de la complejidad de los problemas.

1.4.1 Corrección

Establecer que un algoritmo es correcto implica tres pasos principales. Primero, antes de siquiera intentar determinar si un algoritmo es correcto o no, debemos entender claramente qué significa "correcto". Necesitamos un planteamiento preciso de las características de las entradas con las que se espera que trabaje (llamadas *condiciones previas*) y del resultado que se espera que produzca con cada entrada (las *condiciones posteriores*). Entonces podremos tratar de demostrar enunciados acerca de las relaciones entre las entradas y las salidas, es decir, si se satisfacen las condiciones previas, las condiciones posteriores se cumplirán cuando el algoritmo termine.

Un algoritmo tiene dos aspectos: el método de solución y la sucesión de instrucciones para ponerlo en práctica, es decir, su implementación. Establecer la corrección del método y/o de las

fórmulas empleadas podría ser fácil o podría requerir una larga serie de lemas y teoremas acerca de los objetos con los que el algoritmo trabaja (es decir, grafos, permutaciones, matrices). Por ejemplo, la validez del método de eliminación de Gauss para resolver sistemas de ecuaciones lineales depende de varios teoremas del álgebra lineal. La corrección de algunos de los métodos empleados en algoritmos en este libro no es obvia; se deberán justificar con teoremas.

Una vez establecido el método, lo implementamos en un programa. Si un algoritmo es relativamente corto y directo, por lo regular emplearemos algún medio informal para convencernos de que las distintas partes hacen lo que esperamos que hagan. Podríamos verificar cuidadosamente algunos detalles (por ejemplo, valores iniciales y finales de contadores de ciclos) y simular a mano el algoritmo en unos cuantos ejemplos pequeños. Nada de esto demuestra que el algoritmo es correcto, pero en el caso de programas pequeños podrían bastar las técnicas informales. Se podrían usar técnicas más formales, como las invariantes de ciclo, para verificar la corrección de partes de los programas. En la sección 3.3 veremos más a fondo este tema.

Casi todos los programas que se escriben fuera de clases son muy grandes y complejos. Para demostrar que un programa grande es correcto, podemos tratar de dividirlo en módulos más pequeños; demostrar que, si todos los módulos hacen su trabajo correctamente, el programa total es correcto; y luego demostrar que cada uno de los módulos es correcto. Esta tarea se facilita (aunque sería más correcto decir, "Esta tarea es posible sólo si") los algoritmos y programas se escriben en módulos que en gran medida son independientes y se pueden verificar por separado. Éste es uno de los muchos argumentos de peso en favor de la programación estructurada, modular. Casi todos los algoritmos que presentamos en este libro son los segmentos pequeños con los que se construyen programas grandes, así que no nos ocuparemos de las dificultades que implica demostrar la corrección de algoritmos o programas muy grandes.

No siempre efectuaremos demostraciones formales de corrección en este libro, aunque presentaremos argumentos o explicaciones para justificar las partes complejas o capciosas de los algoritmos. La corrección *puede demostrarse*, aunque en el caso de programas largos y complejos es efectivamente una tarea ardua. En el capítulo 3 presentaremos algunas técnicas que hacen más manejables las demostraciones.

1.4.2 Cantidad de trabajo realizado

¿Cómo mediremos la cantidad de trabajo realizado por un algoritmo? La medida que escojamos deberá ayudarnos a comparar dos algoritmos para el mismo problema de modo que podamos determinar si uno es más eficiente que el otro. Sería bueno que nuestra medida del trabajo nos permitiera comparar aproximadamente los tiempos de ejecución reales de los dos algoritmos, pero no usaremos el tiempo de ejecución como medida del trabajo por varias razones. En primer lugar, dicho tiempo naturalmente varía dependiendo de la computadora empleada, y no nos interesa desarrollar una teoría para una computadora específica. En vez de ello, podríamos contar todas las instrucciones o enunciados ejecutados por un programa, pero esta medida sigue teniendo muchos de los otros defectos del tiempo de ejecución, pues depende en buena medida del lenguaje de programación empleado y del estilo del programador. También sería necesario dedicar tiempo y esfuerzo a escribir y depurar programas para cada algoritmo a estudiar. Queremos una medida del trabajo que nos diga algo acerca de la eficiencia del *método* empleado por un algoritmo, con independencia no sólo de la computadora, el lenguaje de programación y el programador, sino también de los múltiples detalles de implementación, procesamiento fijo (u operaciones de "contabilidad"), como la incrementación de índices de ciclos, el cálculo de índices de arreglos y el es-

tablecimiento de apuntadores en estructuras de datos. Nuestra medida del trabajo deberá ser lo bastante precisa y también lo bastante general como para desarrollar una teoría amplia que sea útil para muchos algoritmos y aplicaciones.

Un algoritmo sencillo podría consistir en algunas instrucciones de inicialización y un ciclo. El número de pasadas por el cuerpo del ciclo es una buena indicación del trabajo efectuado por un algoritmo de ese tipo. Desde luego, la cantidad de trabajo realizada en una pasada por un ciclo podría ser mucho mayor que la efectuada en otra pasada, y un algoritmo podría tener ciclos con un cuerpo más largo que otro, pero ya nos estamos acercando a una buena medida del trabajo. Aunque algunos ciclos tengan, digamos, cinco pasos y otros nueve, si las entradas son grandes el número de pasadas a través de los ciclos generalmente será grande en comparación con el tamaño de los ciclos. Así pues, contar las pasadas a través de todos los ciclos del algoritmo es una buena idea.

En muchos casos, para analizar un algoritmo podemos aislar una operación específica que sea fundamental para el problema que se estudia (o para los tipos de algoritmos en consideración), hacer caso omiso de la inicialización, el control de ciclos y demás tareas de contabilidad, y simplemente contar las operaciones básicas escogidas que el algoritmo efectúa. En muchos algoritmos, se ejecuta exactamente una de estas operaciones cada vez que se pasa por los ciclos principales del algoritmo, así que esta medida es similar a la que describimos en el párrafo anterior.

He aquí algunos ejemplos de operaciones que sería razonable escoger como básicas en varios problemas:

Problema	Operación
Encontrar x en un arreglo de nombres.	Comparación de x con un elemento del arreglo
Multiplicar dos matrices con entradas reales.	Multiplicación de dos números reales (o multiplicación y suma de números reales)
Ordenar un arreglo de números.	Comparación de dos elementos del arreglo
Recorrer un árbol binario (véase la sección 2.3.3).	Recorrer una arista
Cualquier procedimiento no iterativo, incluidos los recursivos.	Invocación de procedimientos

En tanto la o las operaciones básicas se escojan bien y el número total de operaciones efectuadas sea aproximadamente proporcional al número de operaciones básicas, tendremos una buena medida del trabajo realizado por un algoritmo y un buen criterio para comparar varios algoritmos. Ésta es la medida que usaremos en este capítulo y en varios otros del libro. Tal vez el lector no esté totalmente convencido de que ésta es una buena decisión; ampliaremos su justificación en la sección siguiente. Por ahora, nos limitaremos a señalar unos cuantos puntos.

Primero, en algunas situaciones, podría interesarnos intrínsecamente la operación básica: podría ser una operación muy costosa en comparación con las demás, o podría tener algún interés teórico.

Segundo, en muchos casos nos interesa la tasa de crecimiento del tiempo que el algoritmo requiere a medida que el tamaño de las entradas aumenta. En tanto el número total de operaciones sea aproximadamente proporcional al número de operaciones básicas, bastará contar estas últimas para tener una buena idea de qué tan factible es aplicar el algoritmo a entradas grandes.

Por último, escoger esta medida del trabajo nos permite tener mucha flexibilidad. Aunque con frecuencia trataremos de escoger una, o cuando mucho dos, operaciones específicas para contarlas, podríamos incluir algunas operaciones de procesamiento fijo y, en el caso extremo, podríamos escoger como operaciones básicas el conjunto de instrucciones de máquina de una computadora dada. En el otro extremo, podríamos considerar "una pasada por el ciclo" como la operación básica. Así pues, al variar la selección de operaciones básicas, podemos variar el grado de precisión y abstracción de nuestro análisis y así adaptarlo a nuestras necesidades.

¿Qué pasa si escogemos una operación básica para un problema y luego averiguamos que el número total de operaciones efectuadas por un algoritmo no es proporcional al número de operaciones básicas? ¿Y si es considerablemente mayor? En el caso extremo, podríamos escoger una operación básica para cierto problema y luego descubrir que algunos algoritmos que resuelven el problema usan métodos tan distintos que no efectúan *ninguna* de las operaciones que estamos contando. En una situación así, tenemos dos opciones. Podríamos olvidarnos de la operación en cuestión y recurrir al conteo de pasadas a través de ciclos. O bien, si tenemos un interés especial en la operación escogida, podríamos restringir nuestro estudio a una *clase de algoritmos* en particular, una para la que la operación escogida sea apropiada. Los algoritmos que usan otras técnicas y para los que sería apropiado escoger otra operación básica se podrían estudiar aparte. Por lo regular se define una clase de algoritmos para un problema especificando las operaciones que pueden aplicarse a los datos. (El grado de formalidad de las especificaciones variará; en este libro por lo regular serán suficientes las descripciones informales.)

En toda esta sección, hemos usado muchas veces la frase "la cantidad de trabajo efectuada por un algoritmo". Podríamos sustituirla por el término "la complejidad de un algoritmo". *Complejidad* implica la cantidad de trabajo realizada, medida según alguna *medida de complejidad* específica, que en muchos de nuestros ejemplos es el número de operaciones básicas especificadas que se efectúan. Cabe señalar que, en este caso, la complejidad nada tiene que ver con lo complicado o capcioso de un algoritmo; un algoritmo muy complicado podría tener una complejidad baja. Usaremos los términos "complejidad", "cantidad de trabajo efectuada" y "número de operaciones básicas ejecutadas" de forma casi indistinta en este libro.

1.4.3 Análisis promedio y de peor caso

Ahora que tenemos un enfoque general para analizar la cantidad de trabajo efectuada por un algoritmo, necesitamos una forma concisa de presentar los resultados del análisis. La cantidad de trabajo realizado no se puede describir con un solo número porque el número de pasos ejecutados no es el mismo con todas las entradas. Observamos primero que la cantidad de trabajo efectuada generalmente depende del tamaño de las entradas. Por ejemplo, poner en orden alfabético un arreglo de 1000 nombres por lo regular requiere más operaciones que hacerlo con un arreglo de 100 nombres, si se usa el mismo algoritmo. Resolver un sistema de 12 ecuaciones lineales con 12 incógnitas suele requerir más trabajo que resolver un sistema de 2 ecuaciones lineales con 2 incógnitas. Observamos, también, que incluso si consideramos un solo tamaño de entradas, el número de operaciones efectuadas por un algoritmo podría depender de la naturaleza de las entradas. Un algoritmo para poner en orden alfabético un arreglo de nombres podría efectuar muy poco trabajo si sólo unos cuantos nombres no están en orden alfabético, pero podría tener que trabajar mucho más con un arreglo que está totalmente revuelto. Resolver un sistema de 12 ecuaciones lineales podría no requerir mucho trabajo si la mayor parte de los coeficientes consiste en ceros.

La primera observación indica que necesitamos una medida del tamaño de las entradas de un problema. Suele ser fácil escoger una medida razonable del tamaño. He aquí algunos ejemplos:

Problema	Tamaño de las entradas
Encontrar x en un arreglo de nombres.	El número de nombres en el arreglo
Multiplicar dos matrices.	Las dimensiones de las matrices
Ordenar un arreglo de números.	El número de elementos del arreglo
Recorrer un árbol binario.	El número de nodos del árbol
Resolver un sistema de ecuaciones lineales.	El número de ecuaciones, o el número de incógnitas, o ambos
Resolver un problema relativo a un grafo.	El número de nodos del grafo, o el número de aristas, o ambos

Incluso si el tamaño de las entradas es fijo (digamos n), el número de operaciones efectuadas podría depender de las entradas específicas. ¿Cómo, entonces, expresamos los resultados del análisis de un algoritmo? Casi siempre describimos el comportamiento de un algoritmo dando su *complejidad de peor caso*.

Definición 1.10 Complejidad de peor caso

Sea D_n el conjunto de entradas de tamaño n para el problema en consideración, y sea I un elemento de D_n. Sea $t(I)$ el número de operaciones básicas que el algoritmo ejecuta con la entrada I. Definimos la función W así:

$$W(n) = \text{máx } \{t(I) \mid I \in D_n\}.$$

La función $W(n)$ es la *complejidad de peor caso* del algoritmo. $W(n)$ es el número máximo de operaciones básicas que el algoritmo ejecuta con cualquier entrada de tamaño n. ■

En muchos casos es relativamente fácil calcular $W(n)$. En la sección 1.5 presentamos técnicas para casos en los que sería difícil efectuar un cálculo exacto. La complejidad de peor caso es valiosa porque proporciona una cota superior del trabajo efectuado por el algoritmo. El análisis de peor caso podría ayudarnos a estimar un límite de tiempo para una implementación dada de un algoritmo. Efectuaremos un análisis de peor caso para la mayor parte de los algoritmos de este libro. A menos que se diga otra cosa, siempre que nos refiramos a la cantidad de trabajo efectuada por un algoritmo, estaremos hablando de la cantidad de trabajo efectuada en el peor caso.

Podría parecer que una forma más útil y natural de describir el comportamiento de un algoritmo es indicar cuánto trabajo efectúa en promedio; es decir, calcular el número de operaciones ejecutadas con cada entrada de tamaño n y luego sacar el promedio. En la práctica, algunas entradas podrían presentarse con mucha mayor frecuencia que otras, por lo que un promedio ponderado sería más informativo.

Definición 1.11 Complejidad promedio

Sea $Pr(I)$ la probabilidad de que se presente la entrada I. Entonces, el comportamiento promedio del algoritmo se define como

$$A(n) = \sum_{I \in D_n} Pr(I)t(I). \quad \blacksquare$$

Determinamos $t(I)$ analizando el algoritmo, pero $Pr(I)$ no se puede calcular analíticamente. La función $Pr(I)$ se determina por la experiencia y/o con base en información especial acerca de la aplicación que se piensa dar al algoritmo, o haciendo algún supuesto simplificador (por ejemplo que todas las entradas de tamaño n tienen la misma posibilidad de presentarse). Si $Pr(I)$ es complicada, se dificulta el cálculo del comportamiento promedio. Además, claro, si $Pr(I)$ depende de una aplicación dada del algoritmo, la función A describe el comportamiento promedio del algoritmo sólo para esa aplicación.

Los ejemplos que siguen ilustran el análisis de peor caso y promedio.

Ejemplo 1.9 Buscar en un arreglo ordenado

Problema: Sea `E` un arreglo que contiene n entradas (llamadas claves), `E[0], ..., E[n-1]`, sin un orden específico. Encontrar el índice de una clave dada, K, si K está en el arreglo; devolver -1 como respuesta si K no está en el arreglo. (El problema en el que los elementos del arreglo están en orden se estudia en la sección 1.6.)

Estrategia: Comparar K con cada elemento por turno hasta encontrar una coincidencia o hasta agotar el arreglo. Si K no está en el arreglo, el algoritmo devuelve -1 como respuesta.

Hay una clase extensa de procedimientos similares a éste, y los llamamos *rutinas generalizadas de búsqueda*. Es común que sean subrutinas de procedimientos más complejos.

Definición 1.12 Rutina generalizada de búsqueda

Una *rutina generalizada de búsqueda* es un procedimiento que procesa una cantidad indefinida de datos hasta que agota los datos o alcanza su meta. La rutina sigue este patrón de alto nivel:

```
Si no hay más datos que examinar:
        Fracaso.
si no
        Examinar un dato.
        Si este dato es el que buscamos:
            Éxito.
        si no
            Seguir buscando en los datos restantes.
```

El esquema se llama búsqueda *generalizada* porque la rutina a menudo realiza otras operaciones simples mientras busca, como cambiar de lugar elementos de datos, agregar datos a una estructura o quitarle datos, etcétera. ■

Algoritmo 1.1 Búsqueda secuencial, no ordenado

Entradas: E, n, K, donde E es un arreglo con n entradas (indexadas $0, \ldots, n-1$) y K es el elemento buscado. Por sencillez, suponemos que K y los elementos de E son enteros, lo mismo que n.

Salidas: Devuelve `respuesta`, la ubicación de K en E (-1 si no se encuentra K).

```
int BusquedaSec(int[] E, int n, int K)
1.  int respuesta, indice;
2.  respuesta = -1;  // Suponer fracaso.
3.  for (indice = 0; indice < n; indice++)
4.        if (K == E[indice])
5.           respuesta = indice;  // ¡Éxito!
6.           break;  // Tomarse el resto de la tarde libre.
    // Continuar el ciclo.
7.  return respuesta;
```

Operación básica: Comparación de x con un elemento del arreglo.

Análisis de peor caso: Obviamente, $W(n) = n$. Los peores casos se presentan cuando K aparece sólo en la última posición del arreglo y cuando K no está en el arreglo. En ambos casos K se compara con las n entradas.

Análisis de comportamiento promedio: Primero postularemos varios supuestos simplificadores a fin de presentar un ejemplo sencillo, y luego haremos un análisis un poco más complicado con diferentes supuestos. Supondremos que todos los elementos del arreglo son distintos y que, si K está en el arreglo, es igualmente probable que esté en cualquier posición dada.

Como primer caso, supondremos que K está en el arreglo, y denotaremos este suceso con "*éxito*" según la terminología de probabilidades (sección 1.3.2). Las entradas se pueden clasificar según la posición en la que K aparece en el arreglo, de modo que debemos considerar n entradas. Para $0 \leq i < n$, I_i representará el suceso de que K aparece en la i-ésima posición del arreglo. Entonces, sea $t(I)$ el número de comparaciones efectuadas (el número de veces que se prueba la condición de la línea 4) por el algoritmo con la entrada I. Es evidente que, para $0 \leq i < n$, $t(I_i) = i + 1$. Así pues,

$$A_{éxito}(n) = \sum_{i=0}^{n-1} Pr(I_i \mid éxito)\, t(I_i)$$
$$= \sum_{i=0}^{n-1} \left(\frac{1}{n}\right)(i+1) = \left(\frac{1}{n}\right)\frac{n(n+1)}{2} = \frac{n+1}{2}.$$

El subíndice "*éxito*" denota que en este cálculo estamos suponiendo que la búsqueda tuvo éxito. El resultado deberá satisfacer nuestra intuición de que, en promedio, se examinará aproximadamente la mitad del arreglo.

Consideremos ahora el suceso de que K no está en el arreglo, lo que llamaremos "*fracaso*". Sólo hay una entrada para este caso, que llamaremos $I_{fracaso}$. El número de comparaciones en este caso es $t(I_{fracaso}) = n$, así que $A_{fracaso} = n$.

Por último, combinamos los casos en los que K está en el arreglo y no está en el arreglo. Sea q la probabilidad de que K esté en el arreglo. Por la ley de expectativas condicionales (lema 1.2):

$$\begin{aligned} A(n) &= Pr(\acute{e}xito)\, A_{\acute{e}xito}(n) + Pr(fracaso)\, A_{fracaso}(n) \\ &= q\left(\tfrac{1}{2}(n+1)\right) + (1-q)n = n(1-\tfrac{1}{2}q) + \tfrac{1}{2}q. \end{aligned}$$

Si $q = 1$, es decir, si K siempre está en el arreglo, entonces $A(n) = (n + 1)/2$, igual que antes. Si $q = 1/2$, es decir, si hay una posibilidad de 50-50 de que K no esté en el arreglo, entonces $A(n) = 3n/4 + 1/4$; o sea que se examinan aproximadamente las tres cuartas partes del arreglo. Con esto termina el ejemplo 1.9. ■

El ejemplo 1.9 ilustra la interpretación que debemos dar a D_n, el conjunto de entradas de tamaño n. En lugar de considerar todos los posibles arreglos de nombres, números o lo que sea, que podrían presentarse como entradas, identificamos las propiedades de las entradas que afectan el comportamiento del algoritmo; en este caso, si K está o no en el arreglo y, si está, en qué posición. Un elemento I de D_n puede verse como un conjunto (o clase de equivalencia) de todos los arreglos y valores de K tales que K aparece en la posición especificada del arreglo (o no aparece). Entonces $t(I)$ es el número de operaciones efectuadas con cualquiera de las entradas de I.

Observe también que la entrada con la cual el algoritmo tiene su peor comportamiento depende del algoritmo empleado, no del problema. En el caso del algoritmo 1.1, el peor caso se presenta cuando la única posición del arreglo que contiene a K es la última. En el caso de un algoritmo que examina el arreglo de atrás hacia adelante (es decir, comenzando con `indice` = $n - 1$), un peor caso se presentaría si K apareciera sólo en la posición 0. (Otro peor caso sería, una vez más, si K no está en el arreglo.)

Por último, el ejemplo 1.9 ilustra un supuesto que solemos hacer al efectuar un análisis promedio de algoritmos para ordenar o buscar: que los elementos son distintos. El análisis promedio para el caso de elementos distintos da una buena aproximación del comportamiento promedio en casos en que hay pocas repeticiones. Si puede haber muchas repeticiones, es más difícil hacer supuestos razonables acerca de la probabilidad de que la primera aparición de K en el arreglo se dé en alguna posición específica.

Ejemplo 1.10 Multiplicación de matrices

Problema: Sea $A = (a_{ij})$ una matriz de $m \times n$ y $B = (b_{ij})$ una matriz de $n \times p$, ambas con elementos reales. Calcular la matriz producto $C = AB$. (Analizaremos este problema mucho más a fondo en el capítulo 12. En muchos casos suponemos que las matrices son cuadradas, es decir, $m = n$ y $p = n$.)

Estrategia: Usamos el algoritmo que la definición de producto de matrices implica:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad \text{para } 0 \le i < m,\ 0 \le j < p.$$

Algoritmo 1.2 Multiplicación de matrices

Entradas: Matrices A y B, y enteros m, n y p, que indican que A es una matriz de $m \times n$ y B es una matriz de $n \times p$.

Salidas: Matriz C, una matriz de $m \times p$. C se pasa al algoritmo, el cual la llena.

```
multMat(A, B, C, m, n, p)
    for (i = 0; i < m; i++)
        for (j = 0; j < p; j++)
            c_ij = 0;
            for (k = 0; k < n; k++)
                c_ij += a_ik b_kj
```

Operación básica: Multiplicación de elementos de matrices.

Análisis: Para calcular cada elemento de C, se efectúan n multiplicaciones. C tiene mp elementos, así que

$$A(m, n, p) = W(m, n, p) = mnp.$$

En el caso común en el que $m = n = p$, $A(n) = W(n) = n^3$. Con esto termina el ejemplo 1.10. ■

El ejemplo 1.10 ilustra que en el caso de algunos algoritmos, el número de instrucciones ejecutadas, y por ende la cantidad de trabajo efectuada, son independientes de los detalles de las entradas; sólo dependen del tamaño de las entradas. En tales casos, los casos promedio y de peor caso son iguales. En otros algoritmos para el mismo problema, la situación podría ser distinta.

Los conceptos de análisis de comportamiento de peor caso y promedio serían útiles incluso si hubiéramos escogido una medida distinta del trabajo (digamos, tiempo de ejecución). La observación de que la cantidad de trabajo efectuada a menudo depende del tamaño y las propiedades de las entradas llevaría al estudio del comportamiento promedio y el comportamiento de peor caso, sin importar qué medidas se estuvieran usando.

1.4.4 Consumo de espacio

El número de celdas de memoria utilizadas por un programa, al igual que el número de segundos necesarios para ejecutar un programa, depende de la implementación específica. No obstante, es posible sacar algunas conclusiones acerca del consumo de espacio con sólo examinar un algoritmo. Un programa requiere espacio de almacenamiento para sus instrucciones, las constantes y variables que usa, y los datos de entrada. También podría ocupar cierto espacio de trabajo para manipular los datos y guardar información que necesita para efectuar sus cálculos. Los datos de entrada en sí podrían representarse de varias maneras, algunas de las cuales requieren más espacio que otras.

Si los datos de entrada tienen una forma natural (digamos, un arreglo de números o una matriz), entonces analizamos la cantidad de espacio *extra* utilizado, aparte del programa y las entradas. Si la cantidad de espacio extra es constante, sin importar el tamaño de las entradas, decimos que el algoritmo trabaja *en el lugar*. Usamos este término específicamente al hablar de algoritmos de ordenamiento. (Se usa una definición menos estricta de *en el lugar* cuando el espacio extra no es constante, pero sólo es una función logarítmica del tamaño de las entradas, en vista de que la función logaritmo crece muy lentamente; aclararemos todos los casos en los que usemos la definición menos estricta.)

Si hay varias formas de representar las entradas, consideraremos el espacio requerido para las entradas, además del espacio extra que se use, en su caso. En general, hablaremos del número de "celdas" ocupadas sin definir con precisión las celdas. Podríamos suponer que una celda es lo bas-

tante grande como para contener un número o un objeto. Si la cantidad de espacio ocupada depende de las entradas específicas, se podrá efectuar un análisis de peor caso y de caso promedio.

1.4.5 Sencillez

A menudo, pero no siempre, sucede que la forma más sencilla y directa de resolver un problema no es la más eficiente. No obstante, la sencillez es una característica deseable de un algoritmo, pues podría facilitar la verificación de que el algoritmo es correcto, así como la escritura, depuración y modificación de los programas. Al escoger un algoritmo, es recomendable considerar el tiempo necesario para producir un programa depurado, pero si el programa se va a usar con mucha frecuencia, lo más probable es que el factor que determine la selección sea la eficiencia.

1.4.6 Optimidad

Por más ingeniosos que seamos, no podremos mejorar un algoritmo para un problema más allá de cierto punto. Todo problema tiene una complejidad inherente, es decir, existe una cantidad mínima de trabajo que debe efectuarse para resolverlo. Para analizar la complejidad de un problema, no la de un algoritmo específico, escogemos una clase de algoritmos (a menudo especificando los tipos de operaciones que los algoritmos podrán realizar) y una medida de la complejidad, por ejemplo, la o las operaciones básicas que se contarán. Luego, podremos preguntar cuántas operaciones se *necesitan* realmente para resolver el problema. Decimos que un algoritmo es *óptimo* (en el peor caso) si ningún otro algoritmo de la clase estudiada efectúa menos operaciones básicas (en el peor caso). Cabe señalar que, cuando hablamos de los algoritmos de la clase estudiada, no nos referimos únicamente a los algoritmos que han sido ideados. Estamos hablando de todos los algoritmos posibles, incluso los que todavía no se descubren. "Óptimo" no significa "el mejor que se conoce"; significa "el mejor posible".

1.4.7 Cotas inferiores y complejidad de los problemas

Entonces, ¿cómo demostramos que un algoritmo es óptimo? ¿Tenemos que analizar individualmente todos los demás algoritmos posibles (incluso los que ni siquiera se nos han ocurrido)? Afortunadamente, no; podemos demostrar teoremas que establecen una cota o límite inferior del número de operaciones que se necesitan para resolver un problema. Entonces, cualquier algoritmo que ejecute ese número de operaciones será óptimo. Así pues, debemos llevar a cabo dos tareas para encontrar un buen algoritmo o, desde otro punto de vista, para contestar la pregunta: ¿cuánto trabajo es necesario y suficiente para resolver el problema?

1. Idear un algoritmo que parezca eficiente; llamémoslo **A**. Analizar **A** y encontrar una función $W_{\mathbf{A}}$ tal que, con entradas de tamaño n, **A** ejecute cuando más $W_{\mathbf{A}}(n)$ pasos en el peor caso.
2. Para alguna función F, demostrar un teorema que diga que, para cualquier algoritmo de la clase considerada, existe alguna entrada de tamaño n para la cual el algoritmo debe ejecutar al menos $F(n)$ pasos.

Si las funciones $W_{\mathbf{A}}$ y F son iguales, el algoritmo **A** es óptimo (en el peor caso). Si no, podría haber un mejor algoritmo, o podría haber una mejor cota inferior. Cabe señalar que el análisis de un algoritmo específico da una *cota superior* del número de pasos necesarios para resolver un problema, y un teorema del tipo descrito en el inciso 2 da una *cota inferior* del número de pasos necesarios (en el peor caso). En este libro, veremos problemas para los que se conocen algoritmos

óptimos y otros para los que todavía hay una brecha entre la mejor cota inferior conocida y el mejor algoritmo conocido. A continuación presentaremos ejemplos sencillos de cada caso.

El concepto de cota inferior para el comportamiento de peor caso de los algoritmos es muy importante en el campo de la complejidad computacional. El ejemplo 1.11 y los problemas que estudiaremos en la sección 1.6 y en los capítulos 4 y 5 ayudarán a aclarar el significado de las cotas inferiores e ilustrarán técnicas para establecerlas. El lector debe tener presente que la definición "F es una cota inferior para una clase de algoritmos" implica que, para *cualquier* algoritmo de la clase, y cualquier tamaño de entrada n, hay *alguna* entrada de tamaño n con la que el algoritmo debe ejecutar *por lo menos* $F(n)$ operaciones básicas.

Ejemplo 1.11 Hallar el elemento más grande de un arreglo

Problema: Hallar el elemento más grande en un arreglo de n números. (Digamos que el tipo es `float`, para ser específicos, aunque puede ser cualquier tipo numérico.)

Clase de algoritmos: Algoritmos que pueden comparar y copiar números de tipo `float`, pero no efectuar otras operaciones con ellos.

Operación básica: Comparación de un elemento de arreglo con cualquier objeto tipo `float`. Podría ser otro elemento del arreglo o una variable almacenada.

Cota superior: Supóngase que los números están en un arreglo E. El algoritmo siguiente halla el máximo.

Algoritmo 1.3 HallarMáx

Entradas: E, un arreglo de números, definido para los índices $0, \ldots, n-1$; $n \geq 1$, el número de elementos.

Salidas: Devuelve `max`, el elemento más grande de E.

```
int hallarMax(E, n)
1.  max = E[0];
2.  for(indice = 1; indice < n; indice++)
3.      if(max < E[indice])
4.          max = E[indice];
5.  return max;
```

Las comparaciones de elementos del arreglo se efectúan en la línea 3, que se ejecuta exactamente $n - 1$ veces. Por tanto, $n - 1$ es una cota superior del número de comparaciones necesarias para hallar el máximo en el peor caso. ¿Existe algún algoritmo que efectúe menos comparaciones?

Cota inferior: Para establecer una cota inferior podemos suponer que todos los elementos del arreglo son distintos. Este supuesto es permisible porque, si podemos establecer una cota inferior del comportamiento de peor caso para algún subconjunto en entradas (arreglos con elementos distintos), será una cota inferior del comportamiento de peor caso cuando se consideran todas las entradas.

En un arreglo con n elementos distintos, $n - 1$ elementos *no* son el máximo. Podemos concluir que un elemento dado no es el máximo si es menor que por lo menos otro elemento del arreglo. Por tanto, $n - 1$ elementos deberán ser el "perdedor" en comparaciones efectuadas por el algoritmo. Cada comparación tiene sólo un perdedor, de modo que es preciso hacer por lo menos $n - 1$ comparaciones. Es decir, si quedan dos o más "no perdedores" cuando el algoritmo termina, no podrá tener la seguridad de haber identificado el máximo. Por tanto, $F(n) = n - 1$ es una cota inferior del número de comparaciones requeridas.

Conclusión: El algoritmo 1.3 es óptimo. Con esto termina el ejemplo 1.11. ■

Podríamos adoptar un punto de vista un poco diferente para establecer la cota inferior en el ejemplo 1.11. Si se nos proporciona un algoritmo y un arreglo de n números tal que el algoritmo se pare y genere una respuesta después de haber efectuado menos de $n - 1$ comparaciones, podremos demostrar que el algoritmo da la respuesta *equivocada* con cierto conjunto de datos de entrada. Si no se efectúan más de $n - 2$ comparaciones, dos elementos nunca serán perdedores; es decir, no se sabrá que son más pequeños que todos los demás elementos. El algoritmo sólo podrá especificar uno de ellos como el máximo. Bastará con sustituir el otro por un número mayor (si es necesario). Puesto que los resultados de todas las comparaciones efectuadas serán los mismos que antes, el algoritmo dará la misma respuesta que antes, y será errónea.

Este argumento es una demostración por contraposición (véase la sección 1.3.3). Demostramos "si **A** efectúa menos de $n - 1$ comparaciones en cualquier caso, entonces **A** no es correcto". Por contraposición, podemos concluir "si **A** es correcto, **A** efectúa por lo menos $n - 1$ comparaciones en todos los casos". Esto ilustra una técnica útil para establecer cotas inferiores, a saber, demostrar que, si un algoritmo no efectúa suficiente trabajo, es posible acomodar las entradas de modo que el algoritmo dé una respuesta equivocada.

Ejemplo 1.12 Multiplicación de matrices

Problema: Sean $A = (a_{ij})$ y $B = (b_{ij})$ dos matrices de $n \times n$ con elementos reales. Calcular la matriz producto $C = AB$.

Clase de algoritmos: Algoritmos que pueden efectuar multiplicaciones, divisiones, sumas y restas con los elementos de las matrices y con los resultados intermedios que se obtienen al aplicar dichas operaciones a los elementos.

Operación básica: Multiplicación.

Cota superior: El algoritmo acostumbrado (véase el ejemplo 1.10) efectúa n^3 multiplicaciones; por tanto, se requieren cuando más n^3 multiplicaciones.

Cota inferior: Se ha demostrado en la literatura que se requieren por lo menos n^2 multiplicaciones.

Conclusiones: Es imposible saber, con la información disponible, si el algoritmo acostumbrado es o no óptimo. Algunos investigadores han estado tratando de mejorar la cota inferior, es decir, demostrar que se requieren más de n^2 multiplicaciones, mientras que otros han buscado mejores algoritmos. Hasta la fecha, se ha demostrado que el algoritmo acostumbrado *no* es óptimo; existe un método que efectúa aproximadamente $n^{2.376}$ multiplicaciones. ¿Es óptimo ese método? La

cota inferior todavía no se ha mejorado, así que no sabemos si hay algoritmos que efectúan muchas menos multiplicaciones. ■

Hasta aquí hemos estado hablando de cotas inferiores y de optimidad del comportamiento de peor caso. ¿Y el comportamiento promedio? Podemos usar el mismo enfoque que aplicamos al comportamiento de peor caso: escoger un algoritmo que parezca bueno y deducir la función $A(n)$ tal que el algoritmo efectúe $A(n)$ operaciones, en promedio, con entradas de tamaño n; luego, demostrar un teorema que diga que cualquier algoritmo de la clase estudiada debe efectuar por lo menos $G(n)$ operaciones en promedio con entradas de tamaño n. Si $A = G$, podremos decir que el comportamiento promedio del algoritmo es óptimo. Si no, hay que buscar un mejor algoritmo o una mejor cota inferior (o ambas cosas).

En el caso de muchos problemas, analizar con exactitud el número de operaciones es difícil. Se acostumbra considerar un algoritmo como óptimo si el número de operaciones que efectúa está dentro de cierto margen constante respecto al número óptimo exacto (que a su vez sólo suele conocerse dentro de un margen constante). En la sección 1.5 desarrollaremos una metodología para analizar muchos problemas dentro de un margen constante, aunque no podamos efectuar un análisis exacto.

Podemos investigar el consumo de espacio empleando el mismo enfoque que usamos en el análisis de tiempo: analizar un algoritmo dado para obtener una cota superior de la cantidad de espacio que se requiere y demostrar un teorema para establecer una cota inferior. ¿Podemos encontrar para un problema dado un algoritmo que sea óptimo tanto en términos de la cantidad de trabajo efectuado como en la cantidad de espacio usado?, la respuesta a esta pregunta es: a veces. En el caso de algunos problemas, una reducción en el trabajo implica un aumento en el espacio.

1.4.8 Implementación y programación

La *implementación* es la tarea de convertir un algoritmo en un programa de computadora. Los algoritmos podrían describirse con instrucciones detalladas, el tipo de lenguaje de computadora para manipular variables y estructuras de datos, o con explicaciones muy abstractas de alto nivel en lenguaje natural, de métodos para resolver problemas abstractos, sin mencionar representaciones en computadora de los objetos en cuestión. Así pues, la implementación de un algoritmo podría ser una tarea de traducción relativamente directa o podría ser una labor muy larga y difícil que requiere varias decisiones importantes por parte del programador, sobre todo en lo referente a la selección de estructuras de datos. Cuando resulte apropiado, analizaremos la implementación en el sentido general de escoger estructuras de datos y describir formas de ejecutar instrucciones dadas en una descripción en lenguaje natural de un algoritmo. Incluimos este tipo de explicaciones por dos razones. Primera, es una parte natural de importancia el proceso de generar un (buen) programa funcional. Segunda, a menudo es necesario considerar los pormenores de la implementación para analizar un algoritmo; la cantidad de tiempo requerida para efectuar diversas operaciones con objetos abstractos como conjuntos y grafos depende de la forma en que se representan tales objetos. Por ejemplo, formar la unión de dos conjuntos podría requerir sólo una o dos operaciones si los conjuntos se representan como listas ligadas, pero requeriría un gran número de operaciones, proporcional al número de elementos de uno de los conjuntos, si se representan como arreglos y es preciso copiar uno en el otro.

En un sentido estrecho, implementar, o simplemente programar, implica convertir una descripción detallada de un algoritmo y de las estructuras de datos que usa en un programa para una

computadora dada. Nuestro análisis será independiente de la implementación en este sentido; en otras palabras, será independiente de la computadora y el lenguaje de programación empleados y de muchos detalles menores del algoritmo o el programa.

Un programador puede refinar el análisis de los algoritmos considerados utilizando información acerca de la computadora específica que se usará. Por ejemplo, si se cuentan dos o más operaciones, éstas se pueden ponderar según sus tiempos de ejecución; o podría estimarse el número real de segundos que un programa tardará (en el peor caso o el caso promedio). Hay ocasiones en que un conocimiento de la computadora empleada da pie a un nuevo análisis. Por ejemplo, si la computadora tiene ciertas instrucciones potentes poco comunes que se puedan aprovechar eficazmente en el problema a resolver, se podrá estudiar la clase de algoritmos que utilizan esas instrucciones y contarlas como operaciones básicas. Si la computadora tiene un conjunto de instrucciones muy limitado que entorpece la implementación de la operación básica, podría considerarse una clase de algoritmos distinta. No obstante, en general, si se efectuó correctamente el análisis independiente de la implementación, el análisis dependiente del programa deberá servir principalmente para añadir detalles.

Desde luego, también es apropiado efectuar un análisis detallado de la cantidad de espacio que consumen los algoritmos estudiados si se están considerando implementaciones específicas.

Cualquier conocimiento especial acerca de las entradas del problema para el que se busca un algoritmo puede servir para refinar el análisis. Si, por ejemplo, las entradas estarán limitadas a cierto subconjunto de todas las entradas posibles, se podrá efectuar un análisis de peor caso para ese subconjunto. Como ya señalamos, un buen análisis de comportamiento promedio depende de conocer la probabilidad de que se presenten las diversas entradas.

1.5 Clasificación de funciones por su tasa de crecimiento asintótica

¿Qué tan buena es exactamente nuestra medida del trabajo efectuado por un algoritmo? ¿Qué tan precisa es la comparación que podemos efectuar entre dos algoritmos?, puesto que no estamos contando todos y cada uno de los pasos ejecutados por un algoritmo, nuestro análisis tiene por fuerza cierta imprecisión. Hemos dicho que nos conformaremos con que el número total de pasos sea aproximadamente proporcional al número de operaciones básicas contadas. Esto es suficiente para separar algoritmos que efectúan cantidades de trabajo drásticamente distintas con entradas grandes.

Supóngase que un algoritmo para un problema efectúa $2n$ operaciones básicas, y por tanto cerca de $2cn$ operaciones en total, para alguna constante c, y que otro algoritmo efectúa $4.5n$ operaciones básicas, o $4.5c'n$ en total. ¿Cuál se ejecuta en menos tiempo?, en realidad no lo sabemos. El primer algoritmo podría efectuar muchas más operaciones de procesamiento fijo, es decir, su constante de proporcionalidad podría ser mucho mayor. Por tanto, si las funciones que describen el comportamiento de dos algoritmos difieren en un factor constante, podría ser absurdo tratar de distinguirlos (a menos que efectuemos un análisis más refinado). Consideraremos que tales algoritmos pertenecen a la misma clase de complejidad.

Supóngase que un algoritmo para un problema efectúa $n^3/2$ multiplicaciones y otro algoritmo efectúa $5n^2$. ¿Cuál algoritmo se ejecutará en menos tiempo? Si el valor de n es pequeño, el primero hará menos multiplicaciones, pero si n es grande, el segundo será mejor aunque realice más

operaciones de procesamiento fijo. La tasa de crecimiento de una función cúbica es mucho mayor que la de una función cuadrática, así que la constante de proporcionalidad no importa si n es grande.

Como sugieren estos ejemplos, buscamos una forma de comparar y clasificar funciones que no tome en cuenta los factores constantes y las entradas pequeñas. Llegamos a una clasificación con esas características precisamente si estudiamos la *tasa de crecimiento asintótica*, el *orden asintótico* o, simplemente, el *orden* de las funciones.

¿Es razonable hacer caso omiso de las constantes y de las entradas pequeñas? He aquí una analogía nada técnica, nada matemática, que podría ayudar al lector a entender la forma en que usamos el orden asintótico. Suponemos que el lector está escogiendo una ciudad para vivir en ella y su principal criterio es que tenga un clima muy cálido. Las opciones son El Paso, Texas, y Yuma, Arizona. No hay mucha diferencia de temperatura entre las dos, ¿verdad? Pero suponemos que la decisión es entre tres ciudades: El Paso, Yuma y Anchorage, Alaska. Se descartaría Anchorage de inmediato. Esto es análogo a decir que dos funciones son del mismo orden, y una tercera es de un orden distinto. Conocer el orden nos permite hacer distinciones amplias, podemos eliminar las opciones que son deficientes según nuestro criterio.

Ahora bien, ¿cómo decidir entre El Paso y Yuma (o dos algoritmos cuyo tiempo de ejecución es del mismo orden)? Podríamos consultar los registros de temperatura para averiguar que las temperaturas en una ciudad son en promedio unos cuantos grados más altas que en la otra. Esto podría ser análogo a examinar la constante de dos funciones del mismo orden; en el caso de los algoritmos podría implicar contar todas las operaciones, incluidas las de procesamiento fijo, para obtener una estimación más precisa del tiempo de ejecución. Otro enfoque sería considerar otros criterios, tal vez la disponibilidad de empleos y los atractivos culturales al escoger una ciudad, o la cantidad de espacio extra que se usa, al escoger un algoritmo.

¿Hay algún día en que haga más calor en Anchorage que en El Paso? Claro que sí; podría darse un hermoso día de primavera, inusitadamente cálido en Anchorage, mientras un frente frío está pasando por El Paso. Esto no hace que sea erróneo decir, en general, que Anchorage es mucho más frío que El Paso. En las definiciones que daremos de o grande, theta grande y los demás "conjuntos de orden", se hace caso omiso del comportamiento de las funciones con valores pequeños de n. Hacer caso omiso de algunos argumentos pequeños (el tamaño de las entradas, en el caso de los algoritmos) es análogo a hacer caso omiso de los pocos días en que Anchorage podría ser más cálido que El Paso o Yuma.

1.5.1 Definiciones y notación asintótica

Emplearemos la notación acostumbrada para los números naturales y los números reales.

Definición 1.13 Notación para los números naturales y reales

1. El conjunto de los *números naturales* se denota con $\mathbf{N} = \{0, 1, 2, 3, ...\}$.
2. El conjunto de los enteros positivos se denota con $\mathbf{N}^+ = \{1, 2, 3, ...\}$.
3. El conjunto de los números reales se denota con $\mathbf{R}$.
4. El conjunto de los reales positivos se denota con $\mathbf{R}^+$.
5. El conjunto de los reales *no negativos* se denota con $\mathbf{R}^*$. ■

Sean f y g funciones de $\mathbf{N}$ a $\mathbf{R}^*$. La figura 1.5 describe informalmente los conjuntos que usamos para mostrar las relaciones entre los órdenes de las funciones. Tener en mente la figura y las

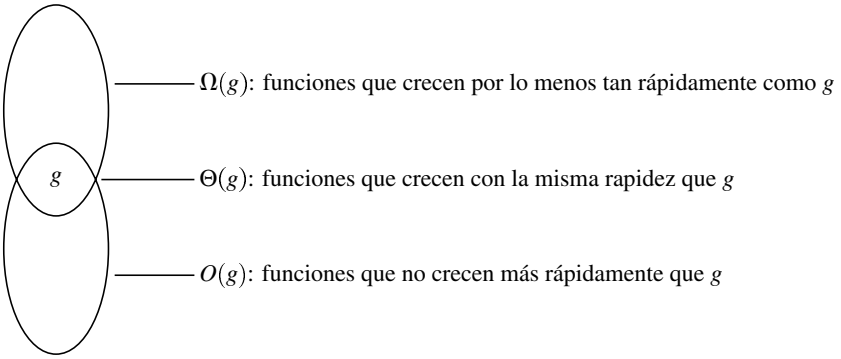


Figura 1.5 Omega grande (Ω), theta grande (Θ) y o grande (O)

definiciones informales ayudará al lector a entender con mayor claridad las definiciones formales y propiedades siguientes.

Definición 1.14 El conjunto $O(g)$

Sea g una función de los enteros no negativos a los números reales positivos. Entonces, $O(g)$ es el conjunto de funciones f, también de los enteros no negativos a los números reales positivos, tal que para alguna constante real $c > 0$ y alguna constante entera no negativa n_0, $f(n) \leq c\, g(n)$ para toda $n \geq n_0$. ■

Con frecuencia es útil pensar en g como alguna función *dada*, y en f como la función que estamos analizando. Observe que una función f podría estar en $O(g)$ aunque $f(n) > g(n)$ para toda n. Lo importante es que f esté acotada por arriba por algún *múltiplo constante* de g. Además, no se considera la relación entre f y g para valores pequeños de n. La figura 1.6 muestra las relaciones de orden de unas cuantas funciones. (Observe que las funciones de la figura 1.6 se trazaron como funciones continuas definidas sobre $\mathbf{R}^+$ o $\mathbf{R}^*$. Las funciones que describen el comportamiento de la mayor parte de los algoritmos que estudiaremos tienen semejantes extensiones naturales.)

El conjunto $O(g)$ suele llamarse "o grande" o simplemente "o de g", aunque la "o" en realidad es la letra griega ómicron. Y, aunque hemos definido $O(g)$ como un conjunto, es común decir "f es o de g" en vez de "f es miembro de o de g".

Hay otra técnica para demostrar que f está en $O(g)$:

Lema 1.5 Una función $f \in O(g)$ si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = c < \infty$, incluido el caso en el que el límite es 0. □

Es decir, si el límite del cociente de f entre g existe y no es ∞, entonces f no crecerá más rápidamente que g. Si el límite es ∞, entonces f sí crece más rápidamente que g.

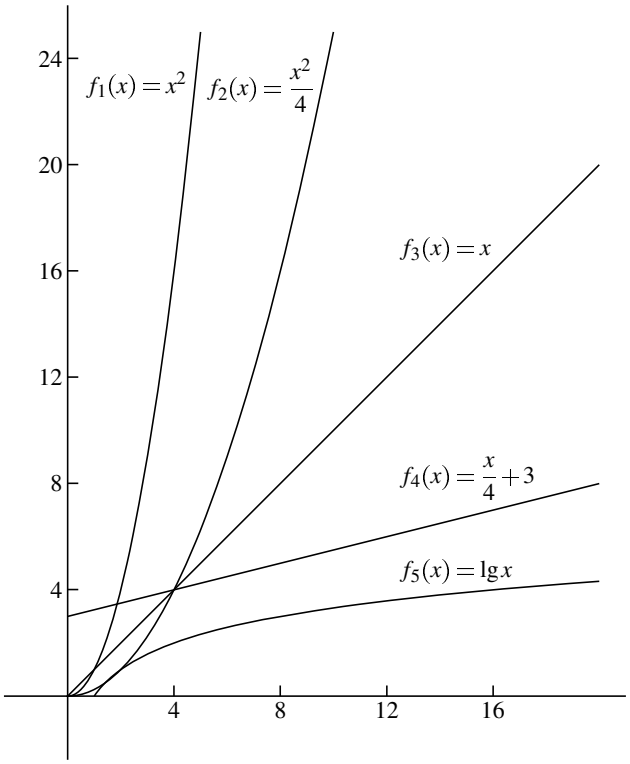


Figura 1.6 Órdenes de funciones: $f_3 \in O(f_4)$, aunque $f_3(x) > f_4(x)$ para $x > 4$, puesto que ambas son lineales. f_1 y f_2 tienen el mismo orden, y crecen con mayor rapidez que las otras tres funciones. f_5 es la función de menor orden de las que se muestran.

Ejemplo 1.13 Funciones con diferente orden asintótico

Sea $f(n) = n^3/2$ y $g(n) = 37n^2 + 120n + 17$. Demostraremos que $g \in O(f)$, pero $f \notin O(g)$.

Puesto que para $n \geq 78$, $g(n) < 1\ f(n)$, se sigue que $g \in O(f)$. Podríamos haber llegado a la misma conclusión de:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{37n^2 + 120n + 17}{n^3/2} = \lim_{n\to\infty} (74/n + 240/n^2 + 34/n^3) = 0.$$

Podemos demostrar que $f \notin O(g)$ observando que el límite de $f/g = \infty$. He aquí un método alterno. Suponemos que $f \in O(g)$ y deducimos una contradicción. Si $f \in O(g)$, entonces existen constantes c y n_0 tales que, para toda $n \geq n_0$,

$$\frac{n^3}{2} \leq 37cn^2 + 120cn + 17c.$$

Así pues,

$$\frac{n}{2} \leq 37c + \frac{120c}{n} + \frac{17c}{n^2} \leq 174c.$$

Puesto que c es una constante y n puede ser arbitrariamente grande, es imposible que $n/2 \leq 174c$ para toda $n \geq n_0$. ■

El teorema que sigue es útil para calcular límites cuando f y g se extienden a funciones continuas, diferenciables, en los reales.

Teorema 1.6 (Regla de L'Hôpital) Sean f y g funciones diferenciables, con derivadas f' y g', respectivamente, tales que

$$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty.$$

Entonces

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}. \quad \square$$

Ejemplo 1.14 Uso de la regla de L'Hôpital

Sea $f(n) = n^2$ y $g(n) = n \lg n$. Demostraremos que $f \notin O(g)$, pero $g \in O(f)$. Primero, simplificamos.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{n \lg n} = \lim_{n\to\infty} \frac{n}{\lg n}.$$

Ahora observamos (véase el lema 1.1) que $\lg n = \ln(n)/\ln(2)$, como preparación para usar la regla de L'Hôpital:

$$\lim_{n\to\infty} \frac{n \ln(2)}{\ln n} = \lim_{n\to\infty} \frac{n \ln(2)}{1/n} = \lim_{n\to\infty} n \ln(2) = \infty.$$

Por tanto, $f \notin O(g)$. Sin embargo, $g \in O(f)$ ya que el cociente inverso tiende a 0. ■

La definición de $\Omega(g)$, el conjunto de funciones que crecen al menos tan rápidamente como g, es el dual de la definición de $O(g)$.[2]

Definición 1.15 El conjunto $\Omega(g)$

Sea g una función de los enternos no negativos a los números reales positivos. Entonces $\Omega(g)$ es el conjunto de funciones f, también de los enteros no negativos a los números reales positivos, tal que, para alguna constante real $c > 0$ y alguna constante entera no negativa n_0, $f(n) \geq c\, g(n)$ para toda $n \geq n_0$. ■

La técnica alterna para demostrar que f está en $\Omega(g)$ es la siguiente:

[2] Los lectores que planeen consultar otros libros y artículos deberán tener en cuenta que la definición de Ω podría ser un poco distinta: la frase "para toda" podría debilitarse a "para infinitamente muchas". La definición de Θ tendrá un cambio acorde.

Lema 1.7 La función $f \in \Omega(g)$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, incluido el caso en el que el límite es ∞. □

Definición 1.16 El conjunto $\Theta(g)$, orden asintótico de g

Sea g una función de los enteros no negativos a los números reales positivos. Entonces $\Theta(g) = O(g) \cap \Omega(g)$, es decir, el conjunto de funciones que están tanto en $O(g)$ como en $\Omega(g)$. La forma más común de leer "$f \in \Theta(g)$" es "f es de orden g". A menudo usamos la frase "orden asintótico" para expresarnos de forma más definitiva, y también se usa el término "complejidad asintótica". ■

También tenemos:

Lema 1.8 La función $f \in \Theta(g)$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, para alguna constante c tal que $0 < c < \infty$. □

Ejemplo 1.15 Orden asintótico de algunos algoritmos

Las complejidades de peor caso del algoritmo 1.1 (búsqueda secuencial, no ordenado) y el algoritmo 1.3 (encontrar el elemento máximo) pertenecen ambas a $\Theta(n)$. La complejidad (peor caso o promedio) del algoritmo 1.2 para multiplicar matrices en el caso $m = n = p$ está en $\Theta(n^3)$. ■

La terminología que suele usarse al hablar acerca de los conjuntos de orden tiene imprecisiones. Por ejemplo, "Éste es un algoritmo de orden n^2" en realidad significa que la función que describe el comportamiento del algoritmo está en $\Theta(n^2)$. En los ejercicios se establecen varios hechos acerca de los conjuntos de orden con que nos topamos comúnmente y de las relaciones entre ellos, como el hecho de que $n(n-1)/2 \in \Theta(n^2)$.

En ocasiones nos interesa indicar que una función tiene un orden asintótico estrictamente más pequeño, o estrictamente más grande que otra. Podemos usar las definiciones siguientes.

Definición 1.17 Los conjuntos $o(g)$ y $\omega(g)$

Sea g una función de los enteros no negativos a los números reales positivos.

1. $o(g)$ es el conjunto de funciones f, también de los enteros no negativos a los números reales positivos, tal que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
2. $\omega(g)$ es el conjunto de funciones f, también de los enteros no negativos a los números reales positivos, tal que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. ■

Por lo regular, "$o(g)$" y "$\omega(g)$" se leen "o pequeña de g" y "omega pequeña de g". Es fácil recordar que las funciones en $o(g)$ son las funciones "más pequeñas" en $O(g)$. Sin embargo, $\omega(g)$ no se usa con mucha frecuencia, ¡probablemente porque es difícil recordar que las funciones en $\omega(g)$ son las funciones *más grandes* de $\Omega(g)$! Se presentan más propiedades de $o(g)$ en los ejercicios 1.33 y 1.34.

Algoritmo	1	2	3	4	
Función de tiempo (μs)	$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	2^n
Tamaño de la entrada (n)	*Tiempo para resolver*				
10	0.00033 s	0.0015 s	0.0013 s	0.0034 s	0.001 s
100	0.003 s	0.03 s	0.13 s	3.4 s	$4 \cdot 10^{16}$ años
1,000	0.033 s	0.45 s	13 s	0.94 h	
10,000	0.33 s	6.1 s	22 min	39 días	
100,000	3.3 s	1.3 min	1.5 días	108 años	
Tiempo permitido	*Tamaño de entrada máximo soluble (aprox.)*				
1 segundo	30,000	2,000	280	67	20
1 minuto	1,800,000	82,000	2,200	260	26

Tabla 1.1 Cómo crecen las funciones.

1.5.2 ¿Cuál es la importacia del orden asintótico?

La tabla 1.1[3] muestra los tiempos de ejecución de varios algoritmos reales para el mismo problema. (La última columna no corresponde a un algoritmo para el problema; se incluye para ilustrar la rapidez con que crecen las funciones exponenciales, y por tanto lo malos que son los algoritmos exponenciales.) Examine los elementos de la tabla para ver con qué rapidez aumenta el tiempo de ejecución al aumentar el tamaño de las entradas en el caso de los algoritmos más complejos. Una de las lecciones importantes de la tabla es que los altos valores constantes de los algoritmos $\Theta(n)$ y $\Theta(n \log n)$ no los hacen más lentos que otros algoritmos a menos que las entradas sean muy pequeñas.

La segunda parte de la tabla examina el efecto de la tasa de crecimiento asintótica sobre el incremento en el tamaño de las entradas que se pueden manejar con más tiempo de computadora (o usando una computadora más rápida). En general *no* se cumple que si multiplicamos el tiempo (o la velocidad) por 60 podremos manejar entradas 60 veces más grandes; eso sólo es cierto en el caso de algoritmos cuya complejidad está en $O(n)$. El algoritmo $\Theta(n^2)$, por ejemplo, puede manejar entradas sólo $\sqrt{60}$ veces más grandes.

Para subrayar aún más el hecho de que el orden asintótico del tiempo de ejecución de un algoritmo es más importante que un factor constante (con entradas grandes), examine la tabla 1.2. Se escribió un programa para el algoritmo cúbico de la tabla 1.1 en la supercomputadora Cray-1; su ejecución tardó $3n^3$ nanosegundos con entradas de tamaño n. El algoritmo lineal se programó en una TRS-80 (una computadora personal económica de los años ochenta); su ejecución tardó $19.5n$ milisegundos (o sea, $19{,}500{,}000n$ nanosegundos). Aunque la constante del algoritmo lineal es 6.5 millones de veces más grande que la constante del algoritmo cúbico, el algoritmo lineal es

[3]Esta tabla (con excepción de la última columna) y la tabla 1.2 se adaptaron de *Programming Pearls* por Jon Bentley (Addison-Wesley, Reading, Mass., 1986) y se reproducen aquí con autorización.

n	Cray-1 Fortran[a] $3n^3$ nanosegundos	TRS-80 Basic[b] $19{,}500{,}000n$ nanosegundos
10	3 microsegundos	0.2 segundo
100	3 milisegundos	2.0 segundos
1,000	3 segundos	20.0 segundos
2,500	50 segundos	50.0 segundos
10,000	49 minutos	3.2 minutos
1,000,000	95 años	5.4 horas

[a] Cray-1 es una marca comercial de Cray Research, Inc.
[b] TRS-80 es una marca comercial de Tandy Corporation.

Tabla 1.2 El orden asintótico gana al final.

Número de pasos realizados para una entrada de tamaño n $f(n)$	Tamaño máximo posible de la entrada s	Tamaño máximo posible de la entrada para tiempos t o cantidades de tiempo muy superiores s_{nuevo}
$\lg n$	s_1	s_1^t
n	s_2	$t\, s_2$
n^2	s_3	$\sqrt{t}\, s_3$
2^n	s_4	$s_4 + \lg t$

Tabla 1.3 Efecto del aumento en la velocidad de la computadora sobre el tamaño máximo de las entradas.

más rápido si el tamaño de las entradas, n, es de 2,500 o más. (Que unas entradas así sean grandes o pequeñas dependería del contexto del problema.)

Si nos concentramos en el orden asintótico de las funciones (de modo que incluimos, por ejemplo, n y 1,000,000 n en la misma clase), entonces, si podemos demostrar que dos funciones *no* son del mismo orden, estaremos haciendo una declaración categórica acerca de la diferencia entre los algoritmos descritos por esas funciones. Si dos funciones *son* del mismo orden, podrían diferir mucho en su factor constante. Sin embargo, el valor de la constante no importa para determinar el efecto de una computadora más rápida sobre el tamaño máximo de entradas que un algoritmo puede manejar en un tiempo dado. Es decir, el valor de la constante no tiene importancia para el incremento entre las dos últimas filas de la tabla 1.1. Examinemos un poco más de cerca el significado de esas cifras.

Supóngase que establecemos un tiempo fijo (un segundo, un minuto; no importa el lapso específico). Sea s el tamaño máximo de las entradas que un algoritmo dado puede manejar en ese tiempo. Supóngase ahora que damos t veces ese tiempo (o que la velocidad de nuestra computadora aumenta en un factor t, sea porque la tecnología ha mejorado o simplemente porque compramos una máquina más cara). La tabla 13.1 muestra el efecto que esto tiene sobre varias complejidades.

Los valores de la tercera columna se calculan observando que

$$\begin{aligned} f(s_{nuevo}) &= \text{número de pasos después de la aceleración} \\ &= t \cdot (\text{número de pasos antes de la aceleración}) = t\, f(s) \end{aligned}$$

y despejando s_{nuevo} de

$$f(s_{nuevo}) = t\, f(s)$$

para s_{nuevo}.

Ahora bien, si multiplicamos las funciones de la primera columna por alguna constante c, ¡los elementos de la tercera columna no cambian! A esto nos referíamos al decir que la constante carece de importancia en cuanto al efecto de un aumento en el tiempo de computadora (o velocidad) sobre el tamaño máximo de las entradas que un algoritmo puede manejar.

1.5.3 Propiedades de O, Ω y Θ

Los conjuntos de orden tienen varias propiedades útiles. Casi todas las demostraciones se dejan como ejercicios; es fácil deducirlas de las definiciones. Para todas las propiedades, suponemos que f, g, h: $\mathbf{N} \to \mathbf{R}^*$. Es decir, las funciones establecen una correspondencia entre enteros no negativos y reales no negativos.

Lema 1.9 Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$; es decir, O es transitivo. También lo son Ω, Θ, o y ω.

Demostración Sean c_1 y n_1 tales que $f(n) \leq c_1 g(n)$ para toda $n \geq n_1$, y sean c_2 y n_2 tales que $g(n) \leq c_2 h(n)$ para toda $n \geq n_2$. Entonces, para toda $n \geq \text{máx}(n_1, n_2)$, $f(n) \leq c_1 c_2 h(n)$. Así pues, $f \in O(h)$. Las demostraciones para Ω, Θ, o y ω son similares. □

Lema 1.10

1. $f \in O(g)$ si y sólo si $g \in \Omega(f)$.
2. Si $f \in \Theta(g)$, entonces $g \in \Theta(f)$.
3. Θ define una relación de equivalencia sobre las funciones. (En la sección 1.3.1 se explica qué es necesario demostrar.) Cada conjunto $\Theta(f)$ es una clase de equivalencia, que llamamos clase de complejidad.
4. $O(f + g) = O(\text{máx}(f, g))$. Se cumplen ecuaciones similares para Ω y Θ. (Son útiles al analizar algoritmos complejos, donde f y g podrían describir el trabajo efectuado por diferentes partes del algoritmo.) □

Puesto que Θ define una relación de equivalencia, podemos indicar la clase de complejidad de un algoritmo especificando cualquier función de la clase. Por lo regular escogemos el representante más sencillo. Así pues, si el número de pasos que un algoritmo ejecuta se describe con la función $f(n) = n^3/6 + n^2 + 2 \lg n + 12$, simplemente decimos que la complejidad del algoritmo está en $\Theta(n^3)$. Si $f \in \Theta(n)$, decimos que f es lineal; si $f \in \Theta(n^2)$, decimos que f es cuadráti-

ca; y si $f \in \Theta(n^3)$, f es cúbica.[4] $O(1)$ denota el conjunto de funciones acotadas por una constante (con n grande).

He aquí dos teoremas útiles. En las demostraciones se usan las técnicas presentadas en la sección 1.5.1, especialmente la regla de L'Hôpital; se dejan para los ejercicios.

Teorema 1.11 lg n está en $o(n^\alpha)$ para cualquier $\alpha > 0$. Es decir, la función logaritmo crece más lentamente que cualquier potencia positiva de n (incluidas potencias fraccionarias). □

Teorema 1.12 n^k está en $o(2^n)$ para cualquier $k > 0$. Es decir, las potencias de n crecen más lentamente que la función exponencial 2^n. (De hecho, las potencias de n crecen más lentamente que cualquier función exponencial c^n, donde $c > 1$.) □

1.5.4 El orden asintótico de sumatorias comunes

La notación de orden facilita deducir y recordar el orden asintótico de muchas sumatorias que se presentan una y otra vez en el análisis de algoritmos. Algunas de estas sumatorias se definieron en la sección 1.3.2.

Teorema 1.13 Sea d una constante no negativa y sea r una constante positiva distinta de 1.

1. La sumatoria de una *serie polinómica* incrementa el exponente en 1.

   Recuerde que una *serie polinómica* de grado d es una sumatoria de la forma $\sum_{i=1}^{n} i^d$. La regla es que este tipo de sumatorias está en $\Theta(n^{d+1})$.

2. La sumatoria de una *serie geométrica* está en Θ de su término más grande.

   Recuerde que una *serie geométrica* es una sumatoria de la forma $\sum_{i=a}^{b} r^i$.

   La regla es válida sea que $0 < r < 1$ o $r > 1$, pero obviamente no cuando $r = 1$. Los límites a y b no son ambos constantes; por lo regular, el límite superior b es alguna función de n y el límite inferior a es una constante.

3. La sumatoria de una *serie logarítmica* está en Θ(el número de términos multiplicado por el logaritmo del término más grande).

   Una *serie logarítmica* es una sumatoria de la forma $\sum_{i=1}^{n} \log(i)$. La regla dice que este tipo de sumatoria está en $\Theta(n \log(n))$. Recuerde que, al hablar de orden asintótico, la base del logaritmo no importa.

[4] Cabe señalar que los términos *lineal*, *cuadrática* y *cúbica* se emplean de forma un tanto menos estricta aquí que como suelen usarlos los matemáticos.

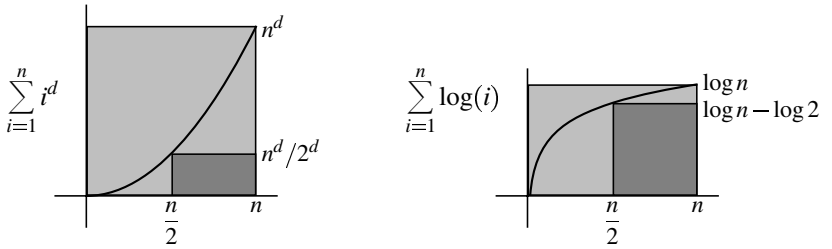


Figura 1.7 Las cotas superior e inferior de muchos tipos de sumatorias están dadas por rectángulos. Si las áreas de los dos rectángulos tienen el mismo orden asintótico, ése deberá ser el orden de la sumatoria.

4. La sumatoria de una *serie polinómica-logarítmica*, que es una sumatoria de la forma $\sum_{i=1}^{n} i^d \log(i)$, está en $\Theta(n^{d+1} \log(n))$.

Demostración Examine la figura 1.7. Puesto que todas las series del teorema tienen la forma $\sum_{i=1}^{n} f(i)$, donde $f(i)$ es monotónica, es evidente que el rectángulo más grande, de altura $f(n)$ y anchura n, es una cota superior de la sumatoria. También, como se ve en la figura 1.4(b), el área bajo la curva de $f(i)$ entre $i = 0$ e $i = n$ es una cota inferior de la sumatoria. En los casos de series polinómicas y series logarítmicas, esa área se puede acotar fácilmente por abajo con el área del rectángulo menor, más oscuro. En la imagen de la izquierda, el área del rectángulo mayor es n^{d+1}, mientras que la del rectángulo menor es $n^{d+1}/2^{d+1}$. Puesto que las áreas de ambos rectángulos tienen el mismo orden asintótico, la sumatoria deberá tener también ese orden. En la imagen de la derecha, las dos áreas son $n \log n$ y $(n/2)(\log n - \log 2)$. Las series polinómicas-logarítmicas son similares, pero esta técnica no funciona para las series geométricas. La regla para las series geométricas es consecuencia directa de la ecuación (1.9) de la sección 1.3.2. □

1.6 Búsqueda en un arreglo ordenado

Para ilustrar las ideas presentadas en las secciones anteriores, estudiaremos un problema conocido.

Problema 1.1 Búsqueda en arreglo ordenado

Dado un arreglo E que contiene n elementos en orden no decreciente, y dado un valor K, encontrar un `indice` para el cual $K =$ `E[indice]` o bien, si K no está en el arreglo, devolver -1 como respuesta. ■

En la práctica, K suele ser la *clave* de un elemento y los elementos pertenecen a alguna clase con otros campos de ejemplar además de la clave, así que un requisito más preciso podría ser $K =$ `E[indice].clave`. Para simplificar la explicación, supondremos que todo el elemento del arreglo es la clave y que es de algún tipo numérico.

Hagamos de cuenta por ahora que no conocemos el algoritmo de Búsqueda Binaria; atacaremos el problema como si fuera la primera vez. Consideraremos varios algoritmos, analizaremos el comportamiento de peor caso y de caso promedio, y al último consideraremos la Búsqueda Binaria y demostraremos que es óptima estableciendo una cota inferior del número de comparaciones de claves que se necesitan.

1.6.1 Algunas soluciones

Observemos que el algoritmo de búsqueda secuencial (algoritmo 1.1) resuelve el problema, pero no utiliza el hecho de que ahora tenemos un arreglo cuyos elementos están en orden. ¿Podemos modificar ese algoritmo de modo que aproveche la información adicional y efectúe menos trabajo?

La primera mejora proviene de la observación de que, dado que el arreglo está en orden no decreciente, tan pronto se llegue a un elemento mayor que K el algoritmo podrá terminar con la respuesta -1. (¿Cómo debe modificarse la prueba de la línea 4 de ese algoritmo para evitar hacer dos comparaciones en cada pasada por el ciclo?) ¿Cómo afecta este cambio el análisis? Es obvio que el algoritmo modificado es mejor en algunos casos y terminará antes con algunas entradas. No obstante, la complejidad de peor caso sigue siendo la misma. Si K es el último elemento del arreglo o si K es mayor que todos los elementos, el algoritmo efectuará n comparaciones.

Para el análisis promedio del algoritmo modificado, debemos saber qué tan factible es que K esté *entre* cualesquier dos elementos del arreglo. Supóngase que definimos un *espacio* g_i, como el conjunto de valores y tales que `E[i-1]` $< y <$ `E[i]` para $i = 1, \ldots, n-1$. También, sea g_0 todos los valores menores que `E[0]` y sea g_n todos los valores mayores que `E[n-1]`. Supondremos, como hicimos en el ejemplo 1.9, que existe una probabilidad q de que K esté en el arreglo. Si K está en `E`, suponemos que todas las posiciones del arreglo son igualmente verosímiles (es decir, tenemos una probabilidad condicional $1/n$). Si K no está en el arreglo, suponemos que todos los espacios son igualmente verosímiles (es decir, tienen una probabilidad condicional de $1/(n+1)$). Para $0 \leq i < n$, se requieren $i+1$ comparaciones para determinar que $K =$ `E[i]` o que K está en g_i, y n comparaciones para determinar que K está en g_n. Así pues, calculamos el número medio de comparaciones, condicionado al éxito ($A_{éxito}$) y condicionado al fracaso ($A_{fracaso}$), como sigue:

$$A_{éxito}(n) = \sum_{i=0}^{n-1} \left(\frac{1}{n}\right)(i+1) = \frac{n+1}{2},$$

$$A_{fracaso}(n) = \sum_{i=0}^{n-1} \left(\frac{1}{n+1}\right)(i+1) + \left(\frac{1}{n+1}\right)n.$$

La primera ecuación corresponde a los casos en que K está en el arreglo, y es la misma que vimos en el ejemplo 1.9. La segunda ecuación corresponde a los casos en que K no está en el arreglo. La sumatoria es fácil de evaluar, y se deja como ejercicio. Al igual que en el ejemplo 1.9, los resultados se combinan en la ecuación $A(n) = qA_{éxito}(n) + (1-q)A_{fracaso}(n)$. El resultado es que $A(n)$ es aproximadamente $n/2$, sin importar el valor de q. El algoritmo 1.1 efectuaba $3n/4$ comparaciones en promedio cuando $q = \frac{1}{2}$, así que el algoritmo modificado es mejor, aunque su comportamiento promedio sigue siendo lineal.

Hagamos otro intento. ¿Podemos encontrar un algoritmo que realice muchas menos de n comparaciones en el peor caso? Supóngase que comparamos K con, digamos, cada cuarto elemen-

to del arreglo. Si hay coincidencia, habremos terminado. Si K es mayor que el elemento con el que se comparó, digamos `E[i]`, no será necesario examinar explícitamente los tres elementos que preceden a `E[i]`. Si $K <$ `E[i]`, entonces K estará entre los dos últimos elementos con los que se comparó. Unas cuantas comparaciones más (¿cuántas?) bastarán para determinar la posición de K si está en el arreglo o para determinar que no está ahí. Los detalles del algoritmo y su análisis se dejan a los lectores, pero es fácil ver que sólo se examinará cerca de la cuarta parte de los elementos del arreglo. Así pues, en el peor caso se efectúan aproximadamente $n/4$ comparaciones.

Podríamos investigar el mismo esquema, escogiendo un valor grande para j y diseñando un algoritmo que compare K con cada j-ésimo elemento, lo que nos permitiría eliminar de la consideración $j - 1$ claves en cada comparación, a medida que avanzamos por el arreglo. Así, efectuaremos aproximadamente n/j comparaciones para encontrar una sección pequeña de E que podría contener a K. Luego efectuamos cerca de j comparaciones para explorar la sección pequeña. Para cualquier j dada, el algoritmo seguirá siendo lineal, pero si escogemos j de modo que $(n/j + j)$ alcance su valor mínimo, encontramos mediante cálculo que el mejor valor de j es $\sqrt{n}$. Entonces, el costo total de la búsqueda será de sólo $2\sqrt{n}$. ¡Hemos roto la barrera del tiempo lineal!

¿Podemos encontrar algo aún mejor? Observemos que nuestra estrategia cambia una vez que encontramos la sección pequeña. Esa sección tiene aproximadamente j elementos, y pagamos j para explorarla, lo cual es un costo lineal. Sin embargo, ahora sabemos que un costo lineal es excesivo. Esto sugiere que debemos aplicar nuestra "estrategia maestra" recursivamente a la sección pequeña, en lugar de cambiar de estrategia.

La idea del tan conocido algoritmo de Búsqueda Binaria lleva aquello de "cada j-ésimo elemento" a su extremo lógico, saltándose la mitad de los elementos en un solo paso. En lugar de escoger un entero específico j y comparar K con cada j-ésimo elemento, comparamos K primero con el elemento que está a la mitad del arreglo. Esto elimina la mitad de las claves con una sola comparación.

Una vez que hemos determinado cuál mitad podría contener a K, aplicamos la misma estrategia recursivamente. Hasta que la sección que podría contener a K se haya encogido hasta tamaño cero, o se haya encontrado a K en el arreglo, seguiremos comparando K con el elemento que está a la mitad de la sección del arreglo que estamos considerando. Después de cada comparación, el tamaño de la sección del arreglo que podría contener a K se recorta a la mitad. Observe que éste es otro ejemplo de rutina de búsqueda generalizada (definición 1.12). El procedimiento *fracasa* cuando la sección que podría contener a K se encoge hasta tamaño cero; *tiene éxito* si encuentra a K, y *sigue buscando* si no ocurre ninguno de esos dos sucesos.

Este procedimiento es un ejemplo sobresaliente del paradigma *divide y vencerás*, que estudiaremos más a fondo en los capítulos 3 y 4. El problema de encontrar a K entre n elementos ordenados se divide en dos subproblemas comparando K con el elemento de la mitad (suponiendo que dicho elemento no sea K). Veremos, mediante análisis, que es más fácil (en el peor caso y en el caso promedio) resolver los dos subproblemas por separado que resolver el problema original sin dividirlo. En realidad, uno de los subproblemas se resuelve con cero trabajo porque sabemos que K no puede estar en esa parte del arreglo.

Algoritmo 1.4 Búsqueda Binaria

Entradas: E, `primero`, `ultimo` y K, donde E es un arreglo ordenado en el intervalo `primero, ..., ultimo` y K es la clave que se busca. Por sencillez, suponemos que k y los elementos de E son enteros, lo mismo que `primero` y `ultimo`.

Salidas: `indice`, tal que `E[indice]` $= K$ si K está en E dentro del intervalo `primero,...,ultimo`, e `indice` $= -1$ si K no está en este intervalo de E.

```
int busquedaBinaria(int[] E, int primero, int ultimo, int K)
1.      if (ultimo < primero)
2.          indice = -1;
3.      else
4.          int medio = (primero + ultimo)/2;
5.          if (K == E[medio])
6.              indice = medio;
7.          else if (K < E[medio])
8.              indice = busquedaBinaria(E, primero, medio-1, K);
9.          else
10.             indice = busquedaBinaria(E, medio+1, ultimo, K);
11.     return indice;
```

La corrección del algoritmo 1.4 se demuestra con todos sus detalles en la sección 3.5.7 como ilustración de una demostración de corrección formal, después de presentar cierto material necesario. El tipo de razonamiento informal que se efectúa más comúnmente se presentó inmediatamente antes del algoritmo.

1.6.2 Análisis en el peor de los casos de la Búsqueda Binaria

Definamos el tamaño del problema de `busquedaBinaria` como $n =$ `ultimo` $-$ `primero` $+ 1$, el número de elementos que hay en el intervalo de E en el que se buscará. Sería razonable escoger como operación básica para el algoritmo de Búsqueda Binaria la comparación de K con un elemento del arreglo. (Una "comparación" aquí siempre se refiere a una comparación con un elemento de E, no una comparación de índices como en la línea 1.) Sea $W(n)$ el número de tales comparaciones que el algoritmo efectúa en el peor caso con arreglos que tienen n entradas en el intervalo a examinar.

Se acostumbra suponer que se efectúa una comparación con una ramificación de tres vías para las pruebas de K de las líneas 5 y 7. (Incluso sin comparaciones de tres vías, se puede lograr aproximadamente la misma cota con comparaciones binarias; véase el ejercicio 1.42.) Así pues, $W(n)$ será también el número de invocaciones de la función `busquedaBinaria`, aparte de la que llega a la línea 2 y sale sin efectuar una comparación.

Detalle de Java: Muchas clases de Java, incluida **`String`**, apoyan comparaciones de tres vías con la interfaz `Comparable`; las clases definidas por el usuario también pueden implementar este recurso; véase el apéndice A.

Supóngase $n > 0$. La tarea del algoritmo consiste en encontrar K en un intervalo de n elementos indizados desde `primero` hasta `ultimo`. El algoritmo llega a la línea 5 y compara K con `E[medio]`, donde `medio` $= \lfloor$(`primero` + `ultimo`)$/2 \rfloor$. En el peor caso estas claves no son iguales y se llega a la línea 8 o a la 10, dependiendo de cuál sección del intervalo, la izquierda o la derecha (relativa a `medio`), podría contener a K. ¿Cuántos elementos hay en estas secciones? Si n es par, habrá $n/2$ elementos en la sección derecha del arreglo y $(n/2) - 1$ en la izquierda; si n es impar, habrá $(n - 1)/2$ elementos en ambas secciones. Por tanto, hay cuando más $\lfloor n/2 \rfloor$ elementos en la sección del arreglo que se especifica en la invocación recursiva. Entonces, considerar que el tamaño del intervalo se reduce a la mitad en cada invocación recursiva es una actitud conservadora.

¿Cuántas veces podemos dividir n entre 2 sin obtener un resultado menor que 1? En otras palabras, ¿qué valor máximo puede tener d sin que deje de cumplirse $n/2^d \geq 1$? Despejamos d: $2^d \leq n$ y $d \leq \lg(n)$. Por tanto, podemos efectuar $\lfloor \lg(n) \rfloor$ comparaciones después de invocaciones recursivas, y una comparación antes de cualquier invocación recursiva; es decir, un total de $W(n) = \lfloor \lg(n) \rfloor + 1$ comparaciones. El ejercicio 1.5 presenta una forma un poco más conveniente de esta expresión, que está bien definida para $n = 0$; se trata de $\lceil \lg(n + 1) \rceil$. Así pues, hemos demostrado que:

Teorema 1.14 El algoritmo de Búsqueda Binaria efectúa $W(n) = \lceil \lg(n + 1) \rceil$ comparaciones de K con elementos del arreglo en el peor caso (donde $n \geq 0$ es el número de elementos del arreglo). Puesto que se efectúa una comparación en cada invocación de la función, el tiempo de ejecución está en $\Theta(\log n)$. □

La Búsqueda Binaria efectúa menos comparaciones en el peor caso que una búsqueda secuencial en el caso promedio.

1.6.3 Análisis de comportamiento promedio

Para simplificar un poco el análisis, supondremos que K aparece en cuando más una posición del arreglo. Como observamos al principio de esta sección, hay $2n + 1$ posiciones que K podría ocupar: las n posiciones de E, a las que llamamos *posiciones de éxito*, y los $n + 1$ espacios, o *posiciones de fracaso*. Para $0 \leq i < n$, I_i representa todas las entradas en las que $K =$ `E[i]`. Para $1 \leq i < n$, I_{n+i} representa las entradas en las que `E[i-1]` $< x <$ `E[i]`. I_n e I_{2n} representan entradas en las que $K <$ `E[0]` y $K >$ `E[n-1]`, respectivamente. Sea $t(I_i)$ el número de comparaciones de K con elementos del arreglo que el algoritmo 1.4 efectúa si se aplica a la entrada I_i. En la tabla 1.4 se muestran los valores de t para $n = 25$. Observe que la mayor parte de los éxitos y todos los espacios están a una distancia de 1 del peor caso; es decir, casi siempre se requieren de 4 a 5 com-

i	$t(I_i)$
0	4
1	5
2	3
3	4
4	5
5	2
6	4
7	5
8	3
9	5
10	4
11	5
12	1
13	4
14	5
15	3
16	4
17	5
18	2
19	4
20	5
21	3
22	5
23	4
24	5
espacios 25, 28, 31, 38, 41, 44	4
todos los demás espacios	5

Tabla 1.4 Número de comparaciones efectuadas por la Búsqueda Binaria, dependiendo de la ubicación de K, para $n = 25$

paraciones para encontrar K. (Con $n = 31$ encontraríamos que la mayor parte de los éxitos y todos los espacios son exactamente el peor caso.) Entonces, si suponemos que todas las posiciones de éxito son igualmente verosímiles, es razonable esperar que el número de comparaciones efectuadas en promedio sea cercano a $\lg n$. El cálculo del promedio suponiendo que todas las posiciones tienen una probabilidad de $1/51$ de $223/52$, que es aproximadamente 4.37, y $\lg 25 \approx 4.65$.

Puesto que el número medio de comparaciones podría depender de la probabilidad de que la búsqueda tenga éxito, denotemos esa probabilidad con q, y definamos $A_q(n)$ como el número medio de comparaciones que se efectúan cuando la probabilidad de tener éxito es q. Por la ley de las expectativas condicionales (lema 1.2), tenemos que

$$A_q(n) = q\,A_1(n) + (1 - q)\,A_0(n).$$

Por tanto, podemos resolver los casos especiales $A_1(n)$ (éxito seguro) y $A_0(n)$ (fracaso seguro) por separado, y combinarlos para obtener una solución para cualquier q. Observe que A_1 equivale a $A_{éxito}$ y A_0 equivale a $A_{fracaso}$, en la nomenclatura empleada para la búsqueda secuencial.

Deduciremos fórmulas aproximadas para $A_0(n)$ y $A_1(n)$, dados estos supuestos:

1. Todas las posiciones de éxito son igualmente verosímiles: $Pr(I_i \mid éxito) = 1/n$ para $1 \le i \le n$.
2. $n = 2^k - 1$, para algún entero $k \ge 0$.

El último supuesto tiene como objetivo simplificar el análisis. El resultado para todos los valores de n es muy cercano al resultado que obtendremos.

Para $n = 2^k - 1$, es fácil ver que toda búsqueda que fracase hará exactamente k comparaciones, sin importar en qué espacio caiga K. Por tanto, $A_0(n) = \lg(n + 1)$.

La clave para analizar el comportamiento promedio de las búsquedas que tienen éxito es dejar de pensar en cuántas comparaciones se efectúan con una entrada específica, I_i, y pensar más bien en el número de entradas con las cuales se efectúa *un número específico* de comparaciones, digamos t. Para $1 \le t \le k$, sea s_t el número de entradas con las que el algoritmo efectúa t comparaciones.

Por ejemplo, con $n = 25$, $s_3 = 4$ porque se efectuarían tres comparaciones con cada una de las cuatro entradas I_2, I_8, I_{15} e I_{21}.

Es fácil ver que $s_1 = 1 = 2^0$, $s_2 = 2 = 2^1$, $s_3 = 4 = 2^2$ y, en general, $s_t = 2^{t-1}$. Puesto que la probabilidad de cada una de las entradas es $1/n$, la probabilidad de que el algoritmo efectúe t comparaciones no es sino s_t/n, y el promedio es

$$A_1(n) = \sum_{t=1}^{k} t\left(\frac{S_t}{n}\right) = \frac{1}{n}\sum_{t=1}^{k} t\,2^{k-1} = \frac{(k-1)2^k + 1}{n}$$

empleando la ecuación (1.12). (Si no supusiéramos $n = 2^k - 1$, el valor de s_k no seguiría el patrón, y algunos fracasos sólo efectuarían $k - 1$ comparaciones, como en la tabla 1.4 para $n = 25$.) Ahora bien, dado que $n + 1 = 2^k$,

$$A_1(n) = \frac{(k-1)(n+1) + 1}{n} = \lg(n+1) - 1 + O\left(\frac{\log n}{n}\right).$$

Como ya mencionamos, $A_0(n) = \lg(n + 1)$ se cumple para el supuesto de que K no está en el arreglo. Por tanto, hemos demostrado el teorema siguiente.

Teorema 1.15 Búsqueda Binaria (algoritmo 1.4) efectúa aproximadamente $\lg(n+1) - q$ comparaciones en promedio con arreglos que tienen n entradas, donde q es la probabilidad de que la búsqueda tenga éxito, y todas las posiciones de éxito son igualmente verosímiles. □

1.6.4 Optimidad

En la sección anterior partimos de un algoritmo $\Theta(n)$, lo mejoramos a $\Theta(\sqrt{n})$ y luego a $\Theta(\log n)$. ¿Puede haber más mejoras? Incluso si no podemos mejorar el orden asintótico, ¿podemos mejorar el factor constante? El papel del análisis de cotas inferiores es decirnos cuándo es posible contestar negativamente una de estas preguntas, o ambas. Una cota inferior "justa", que coincide con la cota superior de nuestro algoritmo, nos asegura que es imposible encontrar mejoras adicionales.

Demostraremos que el algoritmo de búsqueda binaria es óptimo en la clase de algoritmos que no pueden efectuar con los elementos del arreglo otras operaciones que no sean comparaciones. Estableceremos una cota inferior del número de comparaciones requeridas examinando *árboles de decisión* para los algoritmos de búsqueda de esta clase.[5] Sea **A** uno de esos algoritmos. Un árbol de decisión para **A** y un tamaño de entradas dado n es un árbol binario cuyos nodos se rotulan con números entre 0 y $n-1$ y están acomodados según estas reglas:

1. La raíz del árbol se rotula con el índice del primer elemento del arreglo con el que el algoritmo **A** compara K.
2. Supóngase que el rótulo de un nodo en particular es i. Entonces el rótulo del hijo izquierdo de ese nodo es el índice del elemento con el que el algoritmo comparará K a continuación si $K <$ `E[i]`. El rótulo del hijo derecho es el índice del elemento con el que el algoritmo comparará K a continuación si $K >$ `E[i]`. El nodo no tiene un hijo izquierdo (o derecho) si el algoritmo se para después de comparar K con `E[i]` y descubrir que $K <$ `E[i]` (o $K >$ `E[i]`). No hay rama para el caso $K =$ `E[i]`. Un algoritmo razonable no haría más comparaciones en ese caso.

La clase de algoritmos que se puede modelar con este tipo de árboles de decisión es muy amplia; incluye la búsqueda secuencial y las variaciones que consideramos al principio de esta sección. (Cabe señalar que el algoritmo puede comparar dos elementos del arreglo, pero esto no proporciona información, pues el arreglo ya está ordenado, así que no creamos un nodo en el árbol de decisión para esto.) La figura 1.8 muestra el árbol de decisión para el algoritmo de Búsqueda Binaria con $n = 10$.

Dada una entrada en particular, el algoritmo **A** efectuará las comparaciones que se indican a lo largo de un camino que parte de la raíz de su árbol de decisión. El número de comparaciones de claves efectuadas es el número de nodos del camino. El número de comparaciones efectuadas en el peor caso es el número de nodos de un camino de longitud máxima entre la raíz y una hoja, llamaremos a este número p. Supóngase que el árbol de decisión tiene N nodos. Cada nodo tiene cuando más dos hijos, así que el número de nodos a una distancia dada de la raíz (contando cada

[5] Suponemos que el lector conoce la terminología de los árboles binarios, incluidos términos como *raíz*, *hoja* y *camino*; si no es así, le recomendamos adelantarse a la sección 2.3.3 antes de continuar.

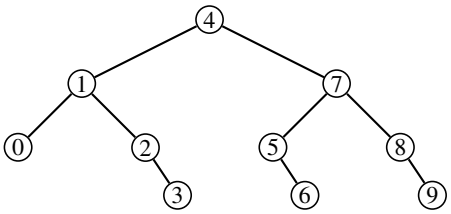


Figura 1.8 Árbol de decisión para al algoritmo de Búsqueda Binaria con $n = 10$

arista como uno) es cuando más dos veces el número que había en la distancia anterior. Puesto que la distancia máxima de la raíz a cualquier nodo es $p - 1$, tenemos

$$N \leq 1 + 2 + 4 + \cdots + 2^{p-1}.$$

Por la ecuación (1.8), el miembro derecho es $2^p - 1$, así que tenemos $2^p \geq (N + 1)$.

Tenemos una relación entre p y N, pero queremos relacionar p con n, el número de elementos del arreglo en el que se buscará. La afirmación clave es que $N \geq n$ si el algoritmo **A** opera correctamente en todos los casos. En particular, decimos que existe en el árbol de decisión algún nodo rotulado i para cada i desde 0 hasta $n - 1$.

Supóngase, por el contrario, que no existe un nodo rotulado i para alguna i dentro del intervalo de 0 a $n - 1$. Podemos formar dos arreglos de entrada `E1` y `E2` tales que `E1[i]` $= K$ pero `E2[i]` $= K' > K$. Para todos los índices j menores que i, hacemos `E1[j]` $=$ `E2[j]` utilizando valores de claves menores que K, ordenados; para todos los índices j mayores que i, hacemos `E1[j]` $=$ `E2[j]` utilizando valores de claves mayores que K', ordenados. Puesto que ningún nodo del árbol de decisión lleva el rótulo i, el algoritmo **A** nunca compara K con `E1[i]` ni con `E2[i]`. **A** se comporta de la misma manera con ambas entradas porque sus demás elementos son idénticos, y debe producir las mismas salidas con ambas entradas. Así pues, **A** produce una salida errónea con al menos uno de los arreglos y por ende no es un algoritmo correcto. Concluimos que el árbol de decisión tiene por lo menos n nodos.

Así pues, $2^p \geq (N + 1) \geq (n + 1)$, donde p es el número de comparaciones en el camino más largo del árbol de decisión. Ahora sacamos logaritmos, y obtenemos $p \geq \lg(n + 1)$. Puesto que **A** era un algoritmo cualquiera de la clase de algoritmos considerado, hemos demostrado el teorema siguiente.

Teorema 1.16 Cualquier algoritmo para encontrar K en un arreglo de n elementos (comparando K con los elementos del arreglo) deberá efectuar por lo menos $\lceil \lg(n + 1) \rceil$ comparaciones con alguna entrada. □

Corolario 1.17 Puesto que el algoritmo 1.4 efectúa $\lceil \lg(n + 1) \rceil$ comparaciones en el peor caso, es óptimo. □

Ejercicios

Sección 1.2 Java como lenguaje algorítmico

1.1 Defina una clase organizadora para información personal que consiste en nombre, dirección, número telefónico y dirección de correo electrónico, haciendo supuestos razonables acerca de cómo sería necesario desglosar estos elementos.

Sección 1.3 Antecedentes matemáticos

1.2 Para toda $n > 0$ y $k > 0$, demostrar que

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{1.32}$$

donde se está usando la notación de la ecuación (1.1). Empleando la notación alterna de esa ecuación, la ecuación (1.32) se convierte en $C(n, k) = C(n - 1, k) + C(n - 1, k - 1)$. Se necesitará el hecho de que $0! = 1$ para algunos casos límite.

1.3 Demuestre la parte 7 del lema 1.1, relativo a logaritmos. *Sugerencia:* Saque los logaritmos de ambos miembros de la ecuación y use la parte 2 de ese lema.

1.4 Demuestre la parte 8 del lema 1.1, relativo a logaritmos.

1.5 Demuestre que $\lceil \lg(n + 1) \rceil = \lfloor \lg n \rfloor + 1$ para enteros $n \geq 1$. *Sugerencia:* Agrupe los valores de n en intervalos de la forma $2^k \leq n \leq 2^{k+1} - 1$.

1.6 Escriba una función (puede ser en pseudocódigo) para obtener $\lceil \lg(n + 1) \rceil$, donde n es un entero no negativo, dividiendo repetidamente n entre 2. Suponga que su lenguaje de programación trunca el resultado de la división entera y desecha el residuo, como hace la mayor parte de los lenguajes. Calcule a mano una tabla de los primeros 10 valores para verificar la función.

1.7

a. ¿Cuántos acomodos distintos hay para una baraja ordinaria de 52 naipes? (Deberá ser fácil encontrar la solución.)

b. Los científicos calculan que han pasado aproximadamente 10^{18} segundos desde el "Big Bang", el inicio del universo. Dé una cota inferior (fácil) para su respuesta a la parte (a) en forma de potencia de 10. Compárela con el número de segundos transcurridos desde el Big Bang.

1.8 Demuestre que si S y T son estocásticamente independientes, entonces

$$Pr(S \mid T) = Pr(S) \quad \text{y} \quad Pr(T \mid S) = Pr(T).$$

1.9 Demuestre, con base en las definiciones, que $Pr(S) = Pr(S \mid T)Pr(T) + Pr(S \mid \text{no } T)Pr(\text{no } T)$.

1.10 ¿Qué probabilidad condicional tienen estos cuatro sucesos dado que $A < B$ y $D < C$, en la situación del ejemplo 1.5: $A < C$, $A < D$, $B < C$, $B < D$?

1.11 En la situación descrita en el ejemplo 1.6, determine $E(I \mid A < D)$ y $E(I \mid D < A)$.

1.12 Suponga que hay tres monedas sobre una mesa. Se escoge una moneda al azar y se lanza. Queremos determinar la probabilidad de que, después del lanzamiento, la mayor parte de las monedas (es decir, dos o tres de ellas) esté con la "cara" hacia arriba, partiendo de diversas configuraciones iniciales. Para cada configuración inicial que se da más adelante, dé nombres a las monedas, defina los sucesos elementales, y dé sus probabilidades. ¿Cuál conjunto de sucesos está definido por la propiedad de que la mayor parte de las monedas muestren "cara" después del lanzamiento, y qué probabilidad tiene este suceso? Suponga que originalmente las monedas muestran

a. cara, sello, sello.
b. sello, sello, sello.
c. cara, cara, sello.

1.13 Considere cuatro dados que contienen los números que se indican a continuación. Para cada par de dados, digamos D_i y D_j, con $1 \leq i, j \leq 4$ e $i \neq j$, calcule la probabilidad de que, en un lanzamiento equitativo de los dos dados, la cara superior de D_i muestre un número mayor que la cara superior de D_j. (Presente los resultados en una matriz de 4×4.)

D_1:	1,	2,	3,	9,	10	11
D_2:	0,	1,	7,	8,	8,	9
D_3:	5,	5,	6,	6,	7,	7
D_4:	3,	4,	4,	5,	11,	12

(Si hace correctamente el cálculo y estudia los resultados con detenimiento, descubrirá que estos dados tienen una propiedad sorprendente. Si usted y otro jugador estuvieran apostando a quién obtiene el número más alto, y usted escogiera su dado primero, el otro jugador siempre podría escoger un dado con alta probabilidad de superar al suyo. Estos dados se comentan en Gardner (1983), donde su descubrimiento se atribuye a B. Efron.)

1.14 Dé una fórmula para $\sum_{i=a}^{n}$, donde a es un entero entre 1 y n.

1.15 Demuestre la ecuación (1.6).

***1.16** Demuestre el lema 1.3. *Sugerencia:* Suponga que $f(x)$ está arriba de la línea de interpolación lineal en *algún* punto entre u y v, habiéndose escogido $u < v$. Entonces, sea w el punto entre u y v tal que $f(w)$ está lo *más arriba* posible sobre la línea en ese intervalo. (Semejante w debe existir si la función es continua.)

***1.17** Demuestre la parte 1 del lema 1.4.

1.18 Demuestre la parte 2 del lema 1.4.

***1.19** Demuestre la parte 3 del lema 1.4. *Sugerencia:* Puede serle útil el lema 1.3.

1.20 Demuestre la ecuación (1.26); es decir, cite las identidades precisas que se necesitan para justificar cada línea de la deducción.

***1.21** Este ejercicio es una oportunidad para efectuar una demostración por contradicción genuina. Demuestra la regla de casos, ecuación (1.31), que se puede plantear así:

Propuesta 1.18 (Regla de casos) Si $(B \Rightarrow C)$ y $(\neg B \Rightarrow C)$, entonces C. □

Comience por suponer $\neg C$, y finalmente deduzca C, utilizando las hipótesis de la propuesta, ecuación (1.27) y *modus ponens* ecuación (1.29).

Sección 1.4 Análisis de algoritmos y problemas

1.22 Dé una fórmula para el número total de operaciones efectuadas por el algoritmo Búsqueda Secuencial (algoritmo 1.1) en el peor caso con un arreglo de n entradas. Cuente las comparacionesde K con elementos del arreglo, las comparaciones con la variable `indice`, las sumas y las asignaciones a `indice`.

1.23 La *mediana* de un conjunto ordenado es un elemento tal que el número de elementos menores que la mediana difiere en cuando más 1 del número de elementos que son mayores, suponiendo que no hay empates.

a. Escriba un algoritmo para hallar la mediana de tres enteros distintos, a, b y c.

b. Describa D, el conjunto de entradas del algoritmo, a la luz de la explicación de la sección 1.4.3 que sigue al ejemplo 1.9.

c. ¿Cuántas comparaciones efectúa su algoritmo en el peor caso? ¿En promedio?

d. ¿Cuántas comparaciones son necesarias en el peor caso para hallar la mediana de tres números? Justifique su respuesta.

1.24 Escriba un algoritmo para hallar el segundo elemento más grande de un conjunto que contiene n elementos. ¿Cuántas comparaciones de elementos efectúa su algoritmo en el peor caso? (Es posible hacer menos de $2n - 3$; consideraremos otra vez este problema más adelante.)

1.25 Escriba un algoritmo para hallar ambos elementos, el más pequeño y el más grande, de un conjunto de n elementos. Trate de encontrar un método que efectúe alrededor de $1.5n$ comparaciones de elementos en el peor caso.

1.26 Dado el polinomio $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, suponga que se usa el algoritmo siguiente para evaluarlo.

```
p = a_0;
potenciax = 1;
for (i = 1; i ≤ n; i++)
      potenciax = x * potenciax;
      p = p + a_i * potenciax;
```

a. ¿Cuántas multiplicaciones se efectúan en el peor caso? ¿Cuántas sumas?
b. ¿Cuántas multiplicaciones se efectúan en promedio?
c. ¿Puede mejorar este algoritmo? (Más adelante volveremos a considerar este problema.)

Sección 1.5 Clasificación de funciones por su tasa de crecimiento asintótica

1.27 Suponga que el algoritmo 1 ejecuta $f(n) = n^2 + 4n$ pasos en el peor caso, y el algoritmo 2 ejecuta $g(n) = 29n + 3$ pasos en el peor caso, con entradas de tamaño n. ¿Con entradas de qué tamaño es más rápido el algoritmo 1 que el algoritmo 2 (en el peor caso)?

1.28 Sea $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ un polinomio en n de grado k con $a_k > 0$. Demuestre que $p(n)$ está en $\Theta(n^k)$.

1.29 Añada una fila a la tabla 1.1 que indique el tamaño máximo aproximado de las entradas que se pueden resolver en un día, para cada columna.

1.30 Sean α y β números reales tales que $0 < \alpha < \beta$. Demuestre que n^α está en $O(n^\beta)$ pero n^β no está en $O(n^\alpha)$.

1.31 Haga una lista de las funciones siguientes, de la de más bajo orden asintótico a la de más alto orden asintótico. Si hay dos (o más) que tengan el mismo orden asintótico, indique cuáles.

a. Comience con estas funciones básicas:

n	2^n	$n \lg n$	n^3
n^2	$\lg n$	$n - n^3 + 7n^5$	$n^2 + \lg n$

***b.** Incorpore las funciones siguientes a su respuesta para la parte (a). Suponga $0 < \epsilon < 1$.

e^n	$\sqrt{n}$	2^{n-1}	$\lg \lg n$
$\ln n$	$(lg\ n)^2$	$n!$	$n^{1+\epsilon}$

***1.32** Demuestre o cite un contraejemplo: Para toda constante positiva c y toda función f de los enteros no negativos a los reales no negativos, $f(cn) \in \Theta(f(n))$. *Sugerencia:* Considere algunas de las funciones de crecimiento rápido de la lista del problema anterior.

***1.33** Demuestre o cite un contraejemplo: Para toda función f de los enteros no negativos a los reales no negativos, $o(f) = O(f) - \Theta(f)$. (Aquí, "$-$" denota diferencia de conjuntos: $A - B$ consiste en los elementos de A que no están en B.)

***1.34** Demuestre o cite un contraejemplo: Para toda función f de los enteros no negativos a los reales no negativos, ninguna función g está al mismo tiempo en $\Theta(f)$ y en $o(f)$, es decir $\Theta(f) \cap o(f) = \varnothing$.

***1.35** Demuestre el lema 1.10.

1.36 Demuestre el teorema 1.11.

***1.37** Demuestre el teorema 1.12.

***1.38** Demuestre que los valores de la tercera columna de la tabla de aceleración (tabla 1.3) no cambian si sustituimos cualquier función $f(n)$ de la primera columna por $cf(n)$, para cualquier constante positiva c.

1.39 Dé un ejemplo de dos funciones, f, g: $\mathbf{N} \rightarrow \mathbf{R}^*$, tales que $f \notin O(g)$ y $g \notin O(f)$.

1.40 Demuestre que se cumple o no se cumple

$$\sum_{i=1}^{n} i^2 \in \Theta(n^2).$$

Sección 1.6 Búsqueda en un arreglo ordenado

1.41 Escriba el algoritmo para buscar K en un arreglo ordenado por el método sugerido en el texto que compara K con cada cuarto elemento hasta hallar K o un elemento mayor que K y luego, en el segundo caso, busca K entre los tres elementos inmediatos anteriores. ¿Cuántas comparaciones efectúa su algoritmo en el peor caso?

1.42 Diseñe una variación de Búsqueda Binaria (algoritmo 1.4) que efectúe sólo una comparación *binaria* (es decir, la comparación devuelve un resultado booleano) de K con un elemento del arreglo cada vez que se invoca la función. Pueden hacerse comparaciones adicionales con variables de intervalo. Analice la corrección de su procedimiento. *Sugerencia:* ¿Cuándo deberá ser de igualdad (==) la única comparación que se hace?

1.43 Dibuje un árbol de decisión para el algoritmo Búsqueda Binaria (algoritmo 1.4) con $n = 17$.

1.44 Describa el árbol de decisión del algoritmo Búsqueda Secuencial (algoritmo 1.1) de la sección 1.4, con cualquier n.

1.45 ¿Cómo podría modificar Búsqueda Binaria (algoritmo 1.4) para eliminar trabajo innecesario si se tiene la certeza de que K está en el arreglo? Dibuje un árbol de decisión para el algoritmo modificado con $n = 7$. Efectúe análisis de comportamiento promedio y de peor caso. (Para el promedio, puede suponerse que $n = 2^k - 1$ para alguna k.)

***1.46** Sea S un conjunto de m enteros. Sea E un arreglo de n enteros distintos ($n \leq m$). Sea K un elemento escogido al azar de S. En promedio, ¿cuántas comparaciones efectuará Búsqueda Binaria (algoritmo 1.4) con E, 0, $n - 1$ y K como entradas? Exprese su respuesta en función de n y m.

***1.47** Las primeras n celdas del arreglo E contienen enteros en orden creciente. El resto de las celdas contiene un entero muy grande que podríamos considerar como infinito (llamémoslo *maxint*). El arreglo podría ser arbitrariamente grande (podemos pensar que es infinito), y *no conocemos n*. Escriba un algoritmo para hallar la posición de un entero dado x ($x < maxint$) en el arreglo en un tiempo $O(\log n)$. (La técnica empleada aquí es útil para ciertos argumentos acerca de los problemas $\mathcal{NP}$-completos que veremos en el capítulo 13.)

Problemas adicionales

***1.48** Es común tener que evaluar la expresión $x \ln(x)$ en $x = 0$, pero $\ln(0) = -\infty$, por lo que no es obvio qué valor debería dársele. Cabe señalar que $-x \ln(x)$ es positivo para $0 < x < 1$. Demuestre que $-x \ln(x)$ se acerca a 0 a medida que x se acerca a 0 desde el lado positivo.

***1.49** Se nos da un espacio de probabilidades con sucesos elementales $U = \{s_1, \ldots, s_k\}$ y algún suceso condicional E. El suceso E define las probabilidades condicionales $p_i(E) = Pr(s_i \mid E)$. (Si $E = U$, entonces $p_i(U) = Pr(s_i)$.)

Defina la *entropía de E* como la función

$$H(E) = -\sum_{i=1}^{k} p_i(E) \lg(p_i(E)). \tag{1.33}$$

El ejercicio 1.48 justifica considerar únicamente los sucesos cuya probabilidad sea distinta de cero, al calcular entropías.

Intuitivamente, la entropía mide la cantidad de *ignorancia* acerca de los sucesos: cuanto mayor es el valor, menos sabemos. Si un suceso es seguro y los demás son imposibles, la entropía es 0. La entropía también puede verse como una medida de la aleatoriedad.

a. Suponga que $Pr(s_i) = 1/k$ para $1 \le i \le k$. Determine $H(U)$.

b. Considere la situación del ejemplo 1.5 para esta parte del ejercicio y todas las siguientes. Por sencillez, use 1.6 como valor de lg(3).

Determine la entropía antes de efectuar cualquier comparación. (Vea la parte a.)

c. Determine la entropía después de descubrir que $A < B$. Es decir, determine $H(E)$, donde E es el suceso $A < B$. ¿Sería diferente si se descubriera que $B < A$?

d. Determine la entropía después de descubrir que $A < B$ y $D < C$. Es decir, determine $H(E)$, donde E es el suceso "$A < B$ y $D < C$". ¿Sería diferente si se invirtiera cualquiera de estas desigualdades?

e. Suponga que el programa compara primero A y B y averigua que $A < B$, y luego compara B y C. Determine la entropía después de cada posible resultado de esta comparación.

f. Suponga que dos programas, P y Q, están tratando de determinar el orden de los elementos. Lo primero que hacen ambos es comparar A y B. Suponga que luego P compara el mayor de A y B con C, igual que en la parte (e), y que Q compara C y D, como en la parte (d). Suponga que ambos programas toman las decisiones óptimas respecto a qué comparar después de las dos primeras comparaciones; *no* es necesario determinar cuáles son las decisiones óptimas.

Con base en esta información, ¿cuál programa *esperaría* usted que necesite efectuar el menor número de comparaciones para determinar el orden total, en el peor caso? ¿Y en el mejor caso? Sólo se pide una conjetura informada y una explicación razonable, no una demostración.

1.50 Usted tiene 70 monedas que supuestamente son de oro y tienen el mismo peso, pero sabe que una de ellas es falsa y pesa menos que las otras. Usted tiene una balanza, y puede colocar cualquier cantidad de monedas en cada plato de la balanza en cada ocasión para saber si los dos lados pesan lo mismo o uno pesa menos que el otro. Bosqueje un algoritmo para encontrar la moneda falsa. ¿Cuántas pesadas efectuará?

Notas y referencias

En la bibliografía hay varios otros textos sobre diseño y análisis de algoritmos.

James Gosling es el principal diseñador de Java. Gosling, Joy y Steele (1996) y ediciones posteriores presenta las especificaciones del lenguaje Java.

Muchas de las referencias que siguen son más avanzadas que este capítulo; sería útil e interesante consultarlas durante toda la lectura de este libro.

El Premio Alan Turing de la ACM se ha otorgado a varias personas que han efectuado trabajos importantes en el campo de la complejidad computacional. Las Turing Award Lectures por Richard M. Karp (1986), Stephen A. Cook (1983) y Michael O. Rabin (1977) presentan tratamientos generales muy amenos de preguntas, técnicas y puntos de vista acerca de la complejidad computacional.

Graham, Knuth y Patashnik (1994) cubren muchas técnicas matemáticas avanzadas útiles. Las ecuaciones (1.11) y (1.19) se dan ahí. Grassmann y Tremblay (1996) presentan una buena introducción a la lógica y las demostraciones.

Knuth (1976) analiza el significado y la historia de las notaciones $O(f)$ y $\Theta(f)$. Brassard (1985) presenta argumentos en favor de la variación de las definiciones empleadas en este libro.

Bentley (1982 y 1986) y sus columnas anteriores "Programming Pearls" en *Communications of the ACM* contienen tratamientos muy amenos del diseño de algoritmos y técnicas para hacer a los programas más eficientes en la práctica.

Los lectores que deseen hojear artículos de investigación encontrarán mucho material en *Journal of the ACM, SIGACT News, SIAM Journal on Computing, Transactions on Mathematical Software* e *IEEE Transactions on Computers*, para mencionar unas cuantas fuentes. Muchas conferencias anuales presentan investigaciones sobre algoritmos.

Recomendamos mucho Knuth (1984), un artículo acerca de la complejidad espacial de las canciones, para cuando la lectura se vuelva ardua.

2

Abstracción de datos y estructuras básicas de datos

2.1 Introducción

La abstracción de datos es una técnica que nos permite concentrarnos en las propiedades importantes de una estructura de datos, sin especificar los aspectos menos importantes. Un *tipo de datos abstracto* (*TDA*) consiste en una declaración de estructura de datos, más un conjunto de operaciones en las que interviene la estructura de datos. El *cliente*, o usuario, de un TDA invoca esas operaciones para crear, destruir, manipular y consultar *objetos* (o *ejemplares*) del tipo de datos abstracto. En este contexto, un *cliente* no es más que algún procedimiento o función definido fuera del TDA.

En este capítulo se describe una técnica para especificar el comportamiento que deben tener los tipos de datos abstractos, se indica cómo aplicar dicha técnica a varias estructuras de datos de amplio uso, y también se analizan algunas propiedades importantes de las estructuras de datos estándar que desempeñarán un papel en el desarrollo posterior de algoritmos.

La técnica de especificación se basa en los trabajos pioneros de David Parnas (véase Notas y Referencias al final del capítulo). La idea clave es el *ocultamiento de la información* o *encapsulamiento de datos*. Los módulos de TDA mantienen datos privados a los que sólo puede accederse desde afuera del módulo a través de operaciones bien definidas. La meta de Parnas era ofrecer una técnica de diseño de software que permitiera trabajar en muchas partes de un proyecto grande de forma independiente, asegurando que las partes encajen y funcionen correctamente.

En el diseño y análisis de algoritmos, los TDA tienen otro papel importante. El diseño principal se puede efectuar utilizando las operaciones de TDA sin decidir cómo se implementarán dichas operaciones. Una vez diseñado el algoritmo en este nivel, podremos efectuar un análisis para contar las veces que el algoritmo usa cada una de las operaciones del TDA. Con esta información, quizá podamos encauzar la implementación de las operaciones del TDA en una dirección tal que las operaciones de uso más frecuente sean las menos costosas.

En otras palabras, podríamos razonar acerca de la *corrección* del algoritmo considerando únicamente las propiedades lógicas de los TDA empleados, que son independientes de la implementación. En cambio, el *análisis de desempeño* depende de la implementación. Diseñar con TDA nos permite separar estos dos intereses.

Un lenguaje de programación *apoya* la abstracción de datos en la medida en que permite al programador restringir el acceso que los clientes tienen a un tipo de datos abstracto; el acceso se restringe a las operaciones definidas y a otras partes públicas de la clase de TDA. El mantenimiento de datos privados se denomina *encapsulamiento* u *ocultamiento de la información*. Esto proporciona al programador una herramienta para asegurar que se conserven ciertas *invariantes* del objeto de TDA. Es decir, si el único acceso de los clientes a un ejemplar del TDA se efectúa a través de un conjunto pequeño de operaciones definidas como interfaz de ese TDA, el programador que implemente las operaciones podrá (al menos en teoría) asegurarse de que las relaciones entre las diferentes partes de la estructura de datos siempre satisfagan las especificaciones del TDA. Dicha situación se sugiere en la figura 2.1. Estas consideraciones explican por qué los TDA son importantes en ingeniería de software.

Escogimos el lenguaje Java para presentar los algoritmos principalmente por la forma sencilla y natural en que apoya la abstracción de datos. En Java, un TDA se identifica como una *clase*. (Sin embargo, no todas las clases son TDA; por ejemplo, vea la sección 1.2.2.) El programa podría crear *objetos* de esta clase, los cuales son simplemente elementos del tipo de datos abstracto.

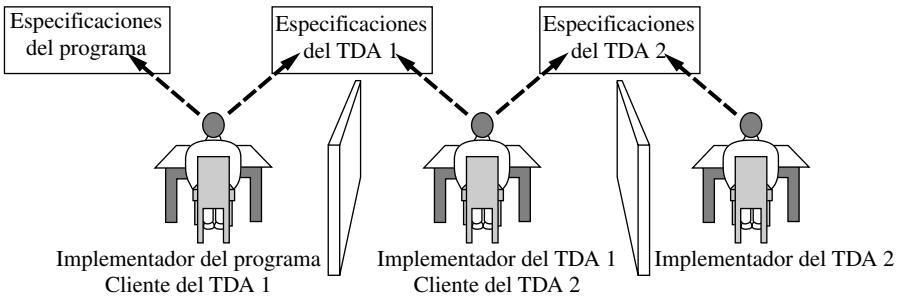


Figura 2.1 Las especificaciones de TDA son la interfaz entre el cliente y el implementador. En este ejemplo, el TDA 1 se implementa utilizando algunos servicios del TDA 2.

2.2 Especificación de TDA y técnicas de diseño

Las *especificaciones* de un TDA describen el comportamiento de las operaciones en términos que tienen sentido para los clientes del TDA. Es decir, las especificaciones deben evitar hacer referencia a campos de ejemplar privados, porque los clientes no tienen conocimiento de ellos. Las especificaciones describen las *relaciones lógicas* entre las partes públicas del TDA, que suelen ser operaciones y constantes. (Ejemplos de especificaciones en secciones posteriores del capítulo aclararán estas generalidades.) Las operaciones de TDA (funciones y procedimientos) se denominan "métodos" en la terminología de Java.

Una ventaja importante de diseñar con TDA es que el cliente puede desarrollar un algoritmo *lógicamente correcto* conociendo únicamente las especificaciones del TDA, sin comprometerse con una implementación específica (ni siquiera con un lenguaje específico) para el TDA. Ésta es la principal justificación para presentar la metodología de TDA en el presente libro.

2.2.1 Especificaciones de TDA

Las especificaciones por lo regular se pueden dividir en *condiciones previas* y *condiciones posteriores*. Las *condiciones previas* de una operación dada son afirmaciones que supuestamente se cumplen en el momento en que se invoca la operación. Si la operación tiene parámetros, es importante que las condiciones previas se planteen en términos de los nombres de esos parámetros, por claridad. Es responsabilidad del cliente satisfacer las condiciones previas antes de invocar cualquier operación (o método estático, o función, o procedimiento) de la clase de TDA. Las *condiciones posteriores* de una operación dada son afirmaciones que el cliente puede suponer que se cumplen en el momento en que la operación termina. Una vez más, si la operación tiene parámetros, es importante plantear las condiciones posteriores en términos de sus nombres. Las condiciones posteriores también se denominan *objetivos* de la operación.

Java ofrece un formato de comentario especial para documentar las clases, lo que incluye las condiciones previas y posteriores de sus métodos. Los comentarios que comienzan con "`/**`" inician un comentario `javadoc`. Usamos la convención de comentarios `javadoc` en el texto para indicar que un comentario tiene que ver con las *especificaciones* de un procedimiento o bloque de código, no con la implementación.

Qué incluye un TDA

Para nuestros fines, un TDA es un conjunto coherente de procedimientos y funciones cuyas especificaciones interactúan para ofrecer cierta capacidad. Adoptaremos una perspectiva minimalista, incluyendo sólo las operaciones necesarias en el TDA en sí; éstas son las operaciones que "necesitan saber" cómo están implementados los objetos. Así pues, un TDA no es una biblioteca de procedimientos que podrían ser útiles; semejantes bibliotecas podrían proporcionarse como clases adicionales, si se necesitan.

Las operaciones necesarias pertenecen a tres categorías: *constructores*, *funciones de acceso* y *procedimientos de manipulación*. Los destructores, que liberan el espacio que ocupa un objeto para que se le pueda dar otro uso, no son cruciales porque Java efectúa "recolección de basura" automática. La recolección de basura encuentra objetos a los que no se hará ya referencia y recicla su espacio.

Definición 2.1 Tipos de operaciones de TDA

Éstas son tres categorías de operaciones para los TDA:

Constructores	crean un objeto nuevo y devuelven una referencia al mismo.
Funciones de acceso	devuelven información acerca de un objeto, pero no lo modifican.
Procedimientos de manipulación	modifican un objeto, pero no devuelven información.

Así pues, una vez creado un objeto, una operación podría modificar el estado del objeto o bien devolver información acerca de su estado, pero no ambas cosas. ■

Cabe señalar aquí que un constructor de TDA no es un constructor en el sentido Java y, al igual que las otras categorías de operaciones de TDA, es independiente del lenguaje de programación. En Java, un constructor *no* debe ir precedido por la palabra clave **`new`**; se usa la misma sintaxis que con cualquier otra función (o método estático).

A causa de nuestra regla de que las funciones de acceso no modifican el estado de ningún objeto, las especificaciones de TDA por lo regular pueden organizarse de forma especial. Normalmente es innecesario especificar condiciones posteriores para las funciones de acceso. Además, al plantear las especificaciones de procedimientos de manipulación y constructores de TDA, sus efectos se deben describir en términos de las funciones de acceso del TDA, en la medida de lo posible. A veces la especificación necesita indicar el efecto combinado de varias operaciones. Al principio podría parecer ilógico examinar la condición posterior de un constructor o procedimiento de manipulación de TDA para averiguar qué "hace" una función de acceso. Sin embargo, si vemos las funciones de acceso colectivamente como una especie de "valor" generalizado de un objeto, este enfoque tiene mucho sentido: siempre que una operación inicializa o altera el estado de un objeto, la condición posterior de esa operación nos deberá decir algo (importante) acerca del nuevo "valor" generalizado del objeto.

Al escoger un conjunto de operaciones para un TDA, es importante asegurarnos que el conjunto de funciones de acceso sea suficiente para verificar las condiciones previas de todas las operaciones. Esto confiere al cliente la capacidad de asegurarse de que ninguna operación se invoque erróneamente.

Para el desarrollo práctico de software, es conveniente contar con una biblioteca de operaciones que se necesitan a menudo con el TDA. La distinción entre la biblioteca y el TDA es que las operaciones de la biblioteca se pueden implementar usando las operaciones del TDA; no es nece-

sario "levantar la tapa del motor" para ver cómo están implementados los objetos. (No obstante, en algunos casos, "levantar la tapa del motor" haría posible una versión más rápida de una función de biblioteca.)

2.2.2 Técnicas de diseño de TDA

En secciones posteriores de este capítulo se presentan las definiciones de varios TDA importantes que usaremos en el desarrollo de algoritmos. Los lectores pueden aprender con ejemplos cómo se usa Java para definir e implementar algunos de esos TDA. También deberá ser fácil entender cómo podrían implementarse en otros lenguajes de programación que tal vez conozcan los lectores.

En el caso de TDA sencillos estándar, como listas ligadas, árboles, pilas y colas FIFO, el TDA que se usa durante el diseño puede usarse también en la implementación final. En algunos casos, otros TDA podrían ser clientes de estos TDA estándar, y usarlos como "bloques de construcción".

En el caso de TDA más complejos o no estándar, como Diccionario, Cola de Prioridad y Unión-Hallar, el TDA se puede usar durante el diseño por sus ventajas lógicas (como simplificar el análisis de corrección), pero en la implementación final podría ser más conveniente "desenvolver" el TDA e implementar un caso especial para el algoritmo que lo usa.

El resto de las secciones de este capítulo presenta varias estructuras de datos estándar y sus tipos de datos abstractos asociados, avanzando en general de lo simple a lo complejo. Se tratarán varios aspectos relacionados con las técnicas de especificación conforme vayan surgiendo. En este capítulo, con excepción de las listas ligadas, sólo se tratarán las implementaciones en algunos de los ejercicios. Incluiremos unos cuantos ejemplos de listas ligadas para proporcionar una muestra de código Java que sirva como guía en otras situaciones. En general, se tratan las implementaciones en los algoritmos que usan los TDA, de modo que la implementación pueda adaptarse al patrón de uso de ese algoritmo.

2.3 TDA elementales: listas y árboles

Los tipos de datos abstractos de listas y árboles son sencillos, pero muy versátiles, cabe indicar que todas sus operaciones se pueden implementar con facilidad en tiempo constante. Especificaremos estos TDA con constructores y funciones de acceso, pero sin procedimientos de manipulación. La ausencia de procedimientos de manipulación hace que las especificaciones sean muy sencillas. En la sección 2.3.2 se explicarán otros motivos para omitir los procedimientos de manipulación. El proceso más natural para definir listas y árboles es en forma recursiva.

2.3.1 TDA recursivos

Un TDA es recursivo si cualquiera de sus *funciones de acceso* devuelve la misma clase del TDA. En otras palabras, alguna parte del objeto (devuelta por la función de acceso) es del mismo tipo que el objeto tratado. En tales casos, el TDA por lo regular también tiene un constructor con un parámetro de la misma clase que el TDA. Un TDA así por fuerza tiene también un constructor no recursivo. Sin embargo, el "constructor no recursivo" suele ser una simple constante (que puede verse como una función que *no* recibe parámetros). Las listas ligadas y los árboles son estructuras de datos comunes que se definen con mayor naturalidad recursivamente. Como veremos en las secciones 2.3.2 a 2.3.5, sus especificaciones son en extremo simples y concisas.

La mejor forma de conceptuar un objeto de un tipo de datos recursivo es como una estructura que incluye no sólo los campos que están inmediatamente accesibles, sino también los campos

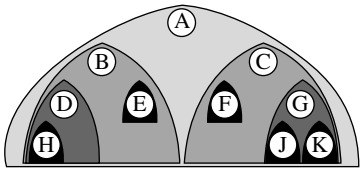


Figura 2.2 Los objetos de un TDA recursivo deben verse como todos los elementos que forman parte transitivamente de la estructura, no sólo el elemento inmediatamente accesible.

que están accesibles *indirectamente* a través de funciones de acceso del TDA, algunas de las cuales devuelven objetos del mismo tipo que el TDA. Por ejemplo, en la figura 2.2 la mejor forma de ver un árbol binario con raíz en *A* es como toda la estructura sombreada, aunque la raíz *A* es el único elemento inmediatamente accesible.

2.3.2 El TDA de lista

Las listas son una estructura de datos fundamental en ciencias de la computación, son de importancia tanto teórica como práctica. Muchos de los algoritmos que desarrollaremos en el presente texto, aunque se presenten empleando arreglos, tienen versiones eficientes en las que las listas son la principal, o la única estructura de datos. El lenguaje de programación *Lisp* se basó originalmente en listas como única estructura de datos del lenguaje; Lisp es un acrónimo de "procesamiento de listas". Varios lenguajes de programación, entre ellos ML y Prolog, incluyen listas como recursos integrados. El TDA de lista que presentamos aquí corresponde a los recursos de listas que ofrecen esos lenguajes, y los nombres de operaciones se han adoptado de Common Lisp.

En este texto, el término *lista* siempre se refiere a lo que suele llamarse *lista ligada* o *lista enlazada* en contextos de estructuras de datos. (En el caso de conjuntos ordenados generales sin pensar en una estructura de datos específica, usaremos el término *sucesión*.) El término más corto *lista* es más apropiado para el TDA porque el término "liga" no aparece en las especificaciones del TDA; si se usan "ligas" en las implementaciones, ese hecho se ocultará a los clientes de `Lista`.

El tipo de lista que más a menudo se necesita en algoritmos, sobre todo en algoritmos para grafos, es una lista de enteros. Por tanto, usaremos esta variedad de lista en los ejemplos ilustrativos de esta sección.

Detalle de Java: Los usuarios experimentados de Java se sentirán tentados a definir `ListaInt`, y las listas de otros tipos de elementos específicos, como subclases de la clase muy general `Lista`. No escogimos esa ruta porque introduce complicaciones cuando los elementos son de un tipo primitivo, y requiere entender perfectamente cómo se toman decisiones de herencia tras bambalinas. Tales temas no son pertinentes para el estudio de algoritmos. Hay muchos textos acerca del lenguaje Java que ahondan en esas posibilidades.

Las especificaciones del TDA `ListaInt` se muestran en la figura 2.3. Como se indica en el pie, las transformaciones a listas de algún otro tipo son directas. Esto es válido también para el código, no sólo los enunciados de especificación. No hay confusión de nombres por tener `cons`, `primero`, `resto` y `nil` en varias clases porque el lenguaje requiere la expresión `ListaInt.cons` para acceder a la versión de la clase `ListaInt`, etcétera.

```
ListaInt cons(int nuevoElemento, ListaInt listaVieja)
```

Condición previa: ninguna.

Condiciones posteriores: Si x = `cons(nuevoElemento, listaVieja)`, entonces:

1. x se refiere a un objeto recién creado;
2. $x \neq$ `nil`;
3. `primero(`x`) = nuevoElemento`;
4. `resto(`x`) = listaVieja`;

```
int primero(ListaInt unaLista)
```

Condición previa: `unaLista` ≠ `nil`.

```
ListaInt resto(ListaInt unaLista)
```

Condición previa: `unaLista` ≠ `nil`.

```
ListaInt nil
```

Constante que denota la lista vacía.

Figura 2.3 Especificaciones del TDA `ListaInt`. La función `cons` es el constructor; `primero` y `resto` son funciones de acceso. El TDA `Lista` es igual, sólo que todas las ocurrencias de `int` se convierten en `Object`, y todas las ocurrencias de `ListaInt` se convierten en `Lista`. Las transformaciones para otros tipos de elementos son similares.

El encabezado del procedimiento, sombreado, indica la rúbrica de tipo de la función o procedimiento en la sintaxis de Java o C. Cada nombre de parámetro va precedido por su tipo. Así, el primer parámetro de `cons` es un `int` y el segundo es un `ListaInt`. El tipo (o clase) que aparece antes del nombre de procedimiento es su tipo devuelto.

El resto de las especificaciones plantea condiciones previas y posteriores. No es necesario decir en las condiciones previas que los parámetros son del tipo apropiado, porque eso está dado en el prototipo. En congruencia con la metodología de la sección 2.2.1, los comportamientos de las funciones de acceso, `primero` y `resto` se describen en la condición posterior de `cons`.

Vale la pena hacer una pausa para pensar en la sencillez del TDA de lista. Es realmente asombroso que todas las funciones computables se puedan calcular empleando listas como única estructura de datos. Hay una constante para la lista vacía y una función (cuyo nombre estándar es `cons`, así que adoptamos ese nombre) para acrecentar una lista colocando un elemento nuevo al principio de una lista anterior (que podría ser la lista vacía). Las demás funciones simplemente devuelven información acerca de una lista (no vacía). ¿Cuál es el primer elemento? ¿Qué lista representa el resto de los elementos? Es evidente que todas las operaciones de `Lista` se pueden implementar en tiempo constante. (Estamos suponiendo que se puede asignar memoria a un objeto nuevo en tiempo constante, lo cual es un supuesto común.)

Las especificaciones de `ListaInt` se pueden implementar de varias maneras sin alterar el código de los *clientes* del TDA. En la figura 2.4 se muestra una implementación representativa (y mínima). Observe que esta implementación no verifica que se satisfagan las condiciones previas

```
import java.lang.*;

public class ListaInt
    {
    int elemento;
    ListaInt siguiente;

    /** La constante nil denota la lista vacía. */
    public static final
    ListaInt nil = null;

    /** Condición previa: L no es nil.
     * Devuelve: primer elemento de L. */
    public static
    int primero(ListaInt L)
        { return L.elemento; }

    /** Condición previa: L no es nil.
     * Devuelve: lista de todos los elementos de L, excepto el 1o. */
    public static
    ListaInt resto(ListaInt L)
        { return L.siguiente; }

    /** Condición previa: ninguna.
     * Condición posterior: sea nuevaL el valor devuelto por cons.
     * Entonces: nuevaL se refiere a un objeto nuevo, nuevaL no es nil,
     * primero(nuevaL) = nuevoElemento, resto(nuevaL) = listaVieja. */
    public static
    ListaInt cons(int nuevoElemento, ListaInt listaVieja)
        {
        Lista nuevaL = new ListaInt();

        nuevaL.elemento = nuevoElemento;
        nuevaL.siguiente = listaVieja;
        return nuevaL;
        }
```

Figura 2.4 Implementación representativa del TDA `ListaInt` como clase Java. Cada objeto tiene los campos de ejemplar privados `elemento` y `siguiente`; el campo público `nil` es una constante debido a la palabra clave **`final`**; las demás partes de la clase son métodos. El programa utilitario `javadoc` asocia un comentario de la forma "`/** ... */`" al elemento de programa que *sigue* al comentario, y da formato a la documentación para navegadores de Web.

de `primero` y `resto`. Es responsabilidad del invocador cuidar que se satisfagan las condiciones previas de cualquier función invocada. Mostramos la implementación de `ListaInt` completa (figura 2.4) como guía para los lectores que desean comenzar a usar Java, además sirve como modelo que otros TDA pueden seguir. En general, no presentaremos código completo en este libro. Los lectores tendrán que completarlo con algunos detalles.

Para fines de ingeniería de software, podría interesarnos crear una clase llamada `ListaInt-Bib`. (No se especificaría constructor para esa clase.) Los métodos de esa biblioteca bien podrían incluir `longitud`, `copiar`, `igual`, `inversa`, `sumatoria`, `max` y `min`.

Detalle de Java: Desde una perspectiva de depuración, podría ser útil incluir el recurso de *error* o *excepción* de Java, pero esto complica la escritura de un conjunto completo de especificaciones tanto para el TDA como para los clientes del TDA. En todo este texto adoptaremos el enfoque de que el código algorítmico se debe concentrar en resolver el problema. Recordamos a los lectores que las consideraciones de ingeniería de software a menudo sugerirán adornos.

Reconstrucción parcial y operaciones no destructivas

Los lectores atentos quizá se estén preguntando cómo *modificamos* una lista bajo el régimen del TDA `ListaInt`. La respuesta es sencilla: ¡No lo hacemos!, no hay *procedimientos de manipulación*. Este TDA es *no destructivo* porque una vez creado un objeto, no es posible actualizarlo. (También se usa el término *inmutable*.) Hay tres posibilidades en el caso de tareas que requieren "actualizar" una lista:

1. En un lenguaje orientado a objetos, como Java, definimos una subclase de `ListaInt` con la capacidad adicional de actualizar (o sea que la subclase sería una clase *destructiva* o *mutable*), o bien
2. Modificamos la clase `ListaInt` misma agregando la capacidad de actualizar (convirtiéndola en una clase *destructiva* o *mutable*), o bien
3. No alteramos la definición de `ListaInt`. Para efectuar una "actualización", reconstruimos parcialmente la lista original para dar una nueva lista y reasignamos la variable de lista de modo que *se refiera a* la nueva lista en vez de la lista original.

La idea de una reconstrucción parcial se ilustra conceptualmente en la figura 2.5. La meta es insertar un elemento nuevo, 22, entre elementos existentes, 13 y 44, de la lista representada por el objeto *w* en la parte superior del diagrama. (Según nuestra explicación anterior acerca de los TDA recursivos, vemos a *w* como toda la lista, no solamente el primer elemento.) Las partes de la lista que contienen los elementos 10 y 13, anteriores al punto de inserción, se "reconstruyen" como se muestra en la parte inferior del diagrama. Desde luego, se crea un objeto nuevo para el elemento 22, pero también se crean los objetos nuevos x' para contener una nueva copia de 13 y w' para contener una nueva copia de 10. Así, los objetos *x* y *w* se mantienen intactos.

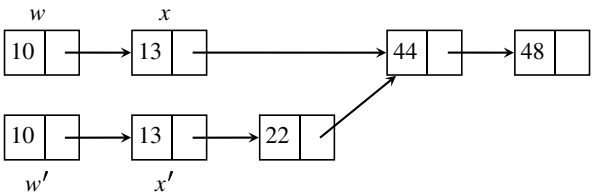


Figura 2.5 Técnica de reconstrucción parcial para insertar 22 en una lista ordenada de 10, 13, 44, 48

En general, una reconstrucción parcial implica que, para cualquier objeto x que tiene un campo que necesitamos modificar, creamos un objeto nuevo x' con valores idénticos en sus demás campos y el valor nuevo en el campo a modificar. No es necesario modificar los objetos a los que tanto x como x' hacen referencia; es por ello que decimos que la reconstrucción es parcial. Pero ahora, si un objeto w que tampoco puede actualizarse hiciera referencia a x y quisiéramos que la "actualización" afecte a w, necesitaríamos reconstruir recursivamente creando w' a partir de w, excepto que w' haría referencia a x', no a x. Como vemos en el ejemplo que sigue, normalmente es fácil localizar a w porque ya usamos w para localizar x, y la invocación de función que usa w sigue activa. Así pues, una vez que volvemos de la invocación de función que creó x', estaremos otra vez en un contexto en el que se conoce w.

Ejemplo 2.1 Inserción en una lista ordenada con reconstrucción parcial

La figura 2.6 muestra el código Java para insertar un entero en una lista ordenada de enteros existente empleando el método de reconstrucción parcial. Al igual que en casi todos los procedimientos recursivos, iniciamos con una prueba para un caso base: ¿`listaVieja` está vacía? Recuerde, ¡la lista vacía está ordenada!, luego probamos otro caso base: ¿Basta con insertar `elementoNuevo` al principio de `listaVieja`? En ambos casos, no será necesario modificar `listaVieja`, e insertamos el nuevo elemento "al principio" con `cons`.

Si no estamos en ninguno de los dos casos base, se hará una invocación recursiva, que devuelve una lista reconstruida (que se almacena en `nuevoResto`) con el nuevo elemento acomodado en su interior. Ahora necesitamos incluir `antiguoPrimero` "al principio" de `nuevoResto`. Puesto que no podemos modificar el objeto `listaVieja`, "reconstruimos" invocando a `cons` para crear un objeto nuevo (que se almacena en `listaNueva`). Observe que `listaNueva` y `listaVieja` tienen el mismo primer elemento en este caso recursivo, pero `resto(listaNueva)` es distinto de `resto(listaVieja)` porque contiene `elementoNuevo` en alguna posición.

Este procedimiento es un ejemplo de *rutina de búsqueda generalizada* (véase la definición 1.12). Estamos "buscando" el elemento delante del cual colocaremos el elemento nuevo, es decir, buscamos un elemento con una clave mayor. El suceso de "fracaso" es la lista vacía, porque obviamente no hay ningún elemento mayor. El suceso de "éxito" es hallar el elemento mayor como primer elemento de la lista que se está examinando. Si no ocurre ninguno de estos sucesos, "seguimos buscando" en el resto de la lista. Se efectúa una operación de reconstrucción por cada paso sin éxito de la búsqueda.

El uso frecuente de variables locales ayuda tanto a depurar como a demostrar corrección. Cabe señalar que las variables locales podrían definirse en "bloques internos" y no tienen que estar al principio de la función. Observe también que, en todos estos ejemplos de código, sólo se asigna un valor a las variables locales una vez en cada invocación de la función; la práctica de asignar un valor y luego sustituirlo por otro valor complica los argumentos de corrección. Trataremos este tema más a fondo en la sección 3.3.

Consideremos el ejemplo de la figura 2.5 en el que se inserta 22 en una lista que contiene 10, 13, 44, 48. La lista inicial es w, con 10 como primer elemento y x como el resto de sus elementos. Puesto que $22 > 10$, es preciso insertar 22 en x, creando una nueva lista x'. Se efectúa una invocación recursiva de `ListaInt.insertar`. Puesto que $22 > 13$, se efectúa una segunda invocación recursiva, la cual crea y devuelve una referencia al nuevo objeto con 22 como primer elemento. Los objetos cuyo primer elemento es 44 y 48 no tuvieron que reconstruirse.

De vuelta en la primera invocación recursiva, se crea un objeto nuevo x' cuyo `resto` es la lista que se acaba de devolver, que comienza con el elemento nuevo 22 y cuyo `primero` se copia

```
/** Condición previa: listaVieja está en orden ascendente.
 * Devuelve: una lista que está en orden ascendente y
 * consiste en elementoNuevo y todos los elementos de listaVieja.
 */
public static
ListaInt insertar1(int elementoNuevo, ListaInt listaVieja)
    {
    ListaInt listaNueva;

    if(listaVieja == ListaInt.nil)
        // elementoNuevo va al principio de listaVieja.
        listaNueva = ListaInt.cons(elementoNuevo, listaVieja);
    else
        {
        int antiguoPrimero = ListaInt.primero(listaVieja);

        if(elementoNuevo ,= antiguoPrimero)
            // elementoNuevo va al principio de listaVieja.
            listaNueva = ListaInt.cons(elementoNuevo, listaVieja);
        else
            {
            ListaInt antiguoResto = ListaInt.resto(listaVieja);
            ListaInt nuevoResto = insertar1(elementoNuevo, antiguoResto);

            // Reconstruir parcialmente listaVieja para dar listaNueva.
            listaNueva = ListaInt.cons(antiguoPrimero, nuevoResto);
            }
        }
    return listaNueva;
    }
```

Figura 2.6 Función (o método Java) para insertar en una lista ordenada de enteros, empleando la técnica de reconstrucción parcial. Observe el uso de calificadores de nombre de clase en los miembros (métodos y campos) de la clase `ListaInt`. Son necesarios porque `insertar1` no pertenece a esa clase.

de x. Este nuevo objeto x' se devuelve (es decir, se devuelve una referencia a él) a la invocación inicial, que conoce a w como lista inicial. La invocación inicial crea w', cuyo `resto` es x' y cuyo `primero` se copia de w, y devuelve una referencia a w' para concluir la operación de inserción en orden. Así, los objetos x' y w' se reconstruyen a partir de x y w conforme nos "salimos" de la recursión.

Ahora que se efectuó la inserción, ¿el programa general seguirá necesitando a w? Obviamente, el TDA de lista no puede contestar esta pregunta. Si la respuesta es "no", el programa general (con toda seguridad) no contendrá referencia alguna a w, porque cualesquier campos o variables que hayan hecho referencia antes a w ahora harán referencia a w'. Si la respuesta es "sí", todavía habrá alguna referencia importante a w en algún lugar del programa. Se sabe que en la práctica las decisiones del programador en cuanto al momento en que se debe liberar y reciclar el espacio

de almacenamiento son fuentes comunes de errores difíciles de localizar, pero un recolector automático de basura ahorra a los programadores tener que tomar tales decisiones. ■

Detalle de Java: Recordamos una vez más a los lectores que conocen C++ que Java no permite al programador definir un nuevo significado para "<=", así que este operador deberá sustituirse por una invocación de método para poder transformar el código de modo que opere con clases no numéricas. Java (a partir de la versión 1.2) ofrece un recurso de *interfaz* llamado `Comparable` para trabajar con clases ordenadas de manera general, como se delinea en el apéndice A.

Se pueden implementar otros TDA utilizando un TDA de lista como bloque de construcción. Ejemplos de ello son los árboles generales (sección 2.3.4) y las pilas (sección 2.4.1). Otros requerirían un TDA de lista actualizable, como los árboles adentro (sección 2.3.5) y las colas (sección 2.4.2).

2.3.3 TDA de árbol binario

Podemos ver los árboles binarios como las generalización no lineal más simple de las listas; en lugar de tener una forma de seguir a otro elemento, hay dos alternativas que llevan a dos elementos distintos. Los árboles binarios tienen muchas aplicaciones en algoritmos.

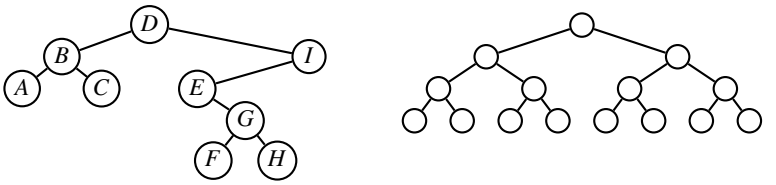
Definiciones y propiedades básicas de los árboles binarios

Matemáticamente, un *árbol binario T* es un conjunto de elementos, llamados nodos, que está vacío o bien satisface lo siguiente:

1. Existe un nodo distinguido *r* llamado *raíz*.
2. Los nodos restantes se dividen en dos subconjuntos disjuntos, *L* y *R*, cada uno de los cuales es un árbol binario. *L* es el *subárbol izquierdo* de *T* y *R* es el *subárbol derecho* de *T*.

Los árboles binarios se representan en papel con diagramas como el de la figura 2.7. Si un nodo *v* es la raíz del árbol binario *T* y *w* es la raíz del subárbol izquierdo (derecho) de *T*, decimos que *w* es el *hijo izquierdo* (*derecho*) de *v* y que *v* es el *padre* de *w*; hay una arista dirigida que va de *v* a *w* en el diagrama. (La dirección es hacia abajo si no hay una punta de flecha.)

El *grado* de un nodo de árbol es el número de subárboles no vacíos que tiene. Un nodo de grado cero es una *hoja*. Los nodos con grado positivo son *nodos internos*.



(a) Con rótulos de nodo (b) Completo

Figura 2.7 Árboles binarios

```
ArbolBin construirArbol(Object nuevaRaiz, ArbolBin viejoAI, ArbolBin viejoAD)
```

Condición previa: ninguna.

Condiciones posteriores: Si x = `construirArbol(nuevaRaiz, viejoAI, viejoAD)`, entonces:

1. x se refiere a un objeto recién creado;
2. $x \neq$ `nil`;
3. `raiz(`x`) = nuevaRaiz`;
4. `subarbolIzq(`x`) = viejoAI`;
5. `subarbolDer(`x`) = viejoAD`;

```
Object raiz(ArbolBin a)
```

Condición previa: `a` $\neq$ `nil`.

```
ArbolBin subarbolIzq(ArbolBin a)
```

Condición previa: `a` $\neq$ `nil.`

```
ArbolBin subarbolDer(ArbolBin a)
```

Condición previa: `a` $\neq$ `nil.`

```
ArbolBin nil
```

Constante que denota el árbol vacío.

Figura 2.8 Especificaciones del TDA `ArbolBin`. La función `construirArbol` es el constructor; `raiz`, `subarbolIzq` y `subarbolDer` son funciones de acceso. Las especializaciones en las que los nodos pertenecen a una clase más específica que **`Object`** se definen de forma análoga.

La *profundidad* de la raíz es 0 y la de cualquier otro nodo es uno más la profundidad de su padre.[1] Un *árbol binario completo* es un árbol binario en el que todos los nodos internos tienen grado 2 y todas las hojas están a la misma profundidad. El árbol binario de la derecha de la figura 2.7 es completo.

La *altura* de un árbol binario (que algunos también llaman su *profundidad*) es el máximo de las profundidades de sus hojas. La *altura* de cualquier nodo de un árbol binario es la altura del subárbol del cual es la raíz. En la figura 2.7(a), la profundidad de *I* es 1 y su altura es 3; la profundidad de *D* es cero y su altura es 4.

Los hechos que siguen se usarán con frecuencia en el texto. Las demostraciones son fáciles y se omiten.

Lema 2.1 Hay cuando más 2^d nodos a una profundidad d en un árbol binario. □

Lema 2.2 Un árbol binario con altura h tiene cuando más $2^{h+1} - 1$ nodos. □

Lema 2.3 La altura de un árbol binario con n nodos es por lo menos $\lceil \lg(n + 1) \rceil - 1$. □

La figura 2.8 da las especificaciones del TDA `ArbolBin`. Son obvias las analogías con el TDA de lista de la sección 2.3.2. La función de acceso `raiz` es análoga a `Lista.primero`; ac-

[1]Cuidado: algunos autores definen *profundidad* de modo que la profundidad de la raíz es 1.

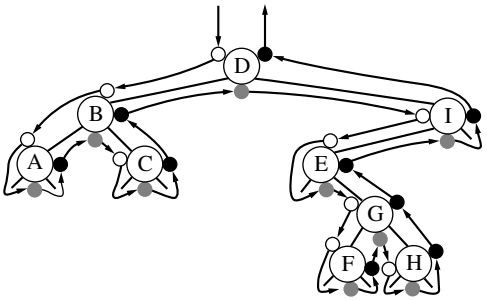


Figura 2.9 Recorrido de un árbol binario como viaje alrededor del árbol

cede al dato que está inmediatamente disponible. Sin embargo, en lugar de `Lista.resto` hay dos funciones de acceso, `subarbolIzq` y `subarbolDer`, que permiten al cliente acceder a sólo una parte del resto del árbol.[2]

Recorrido de un árbol binario

Podemos pensar en un recorrido estándar de un árbol binario como un viaje en lancha alrededor del árbol, partiendo de la raíz, como sugiere la figura 2.9. Imaginamos que cada nodo es una isla y cada arista es un puente tan bajo que la lancha no puede pasar por debajo. (Para que la imagen funcione correctamente, también imaginamos que sobresale un muelle en todos los lugares donde hay un árbol vacío.) La lancha parte del nodo raíz y navega a lo largo de las aristas, visitando nodos en su camino. La primera vez que se visita un nodo (punto blanco) es su vez de *orden previo*, la segunda vez que se visita (punto gris, al regresar del hijo izquierdo) es su vez de *orden interno*, y la última vez que se visita (punto negro, al regresar del hijo derecho) es su vez de *orden posterior*. El recorrido de árboles se puede expresar de forma elegante como procedimiento recursivo, con el siguiente esqueleto:

```
void recorrer(ArbolBin T)
     if(T no está vacío)
        Procesar-orden previo raiz(T);
        recorrer(subarbolIzq(T));
        Procesar-orden interno raiz(T);
        recorrer(subarbolDer(T));
        Procesar-orden posterior raiz(T);
     return;
```

El tipo devuelto de `recorrer` variará dependiendo de la aplicación, y también podría recibir parámetros adicionales. El procedimiento anterior muestra el esqueleto común.

[2] Estos nombres no son estándar, y en otras obras se usan los nombres "`hijoIzq`" e "`hijoDer`". No obstante, en el contexto del TDA, es mejor ver el objeto como todo el subárbol, no sólo su nodo raíz. En nuestra terminología, los hijos izquierdo y derecho son las *raíces de* los subárboles izquierdo y derecho, respectivamente.

Para el árbol binario de la figura 2.9, los órdenes de recorrido de los nodos son los siguientes:

Orden previo (puntos blancos):	D	B	A	C	I	E	G	F	H
Orden interno (puntos grises):	A	B	C	D	E	F	G	H	I
Orden posterior (puntos negros):	A	C	B	F	H	G	E	I	D

2.3.4 El TDA de árbol

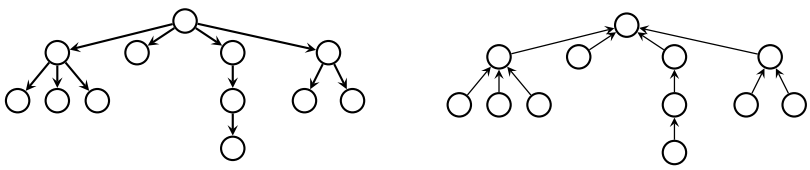
Un *árbol general* (en términos más precisos, un *árbol afuera general*) es una estructura no vacía con nodos y aristas dirigidas tales que un nodo, la *raíz*, no tiene aristas que lleguen a él y todos los demás nodos tienen exactamente una arista que llega a ellos. Además, existe un camino desde la raíz a todos los demás nodos. No hay restricción respecto al número de aristas que salen de cualquier nodo. Un *bosque* es una colección de árboles individuales.

Cada nodo de un árbol es la raíz de su propio *subárbol*, que consiste en todos los nodos a los que puede llegar, incluido él mismo. Decimos que cada arista va del padre al hijo. Si el nodo v es padre del nodo w en un árbol, el árbol cuya raíz es w es un *subárbol principal* del árbol cuya raíz es v. Cada subárbol principal de un árbol tiene menos nodos que todo el árbol. No es factible nombrar individualmente cada subárbol principal, por lo que el TDA `Arbol` es un poco más complejo que el TDA `ArbolBin`.

En un árbol general, los subárboles no tienen por fuerza un orden inherente, mientras que en un árbol binario tienen el orden "izquierdo" y "derecho". (Si consideramos que los subárboles de un árbol general *sí* están ordenados, la estructura se denomina "árbol ordenado".) Otra diferencia respecto a los árboles binarios es que no existe una representación para un árbol general vacío.

Si todas las aristas están orientadas hacia la raíz en vez de alejarse de la raíz, la estructura es un *árbol adentro*, y las aristas van del hijo al padre (véase la figura 2.10). Las estructuras de datos y operaciones apropiadas para esta variedad de árboles son diferentes, como veremos en la sección 2.3.5.

El TDA `Arbol` (una vez más, con una colección mínima de operaciones) se describe con las especificaciones de la figura 2.11. Son obvias las similitudes con el TDA `ArbolBin`. Sin embargo, en lugar de dos subárboles con nombre, tenemos un número indefinido de *subárboles principales*, por lo que `Lista` es la estructura natural para tales objetos. A menos que se considere que el árbol está ordenado, el orden que la lista imparte es incidental, y los subárboles se manejan como un conjunto, no como una sucesión.



(a) árbol afuera (b) árbol adentro

Figura 2.10 Árbol afuera general y el árbol adentro correspondiente

```
Arbol construirArbol(Object nuevaRaiz, ListaArboles viejosArboles)
```

Condición previa: ninguna.

Condiciones posteriores: Si x = `construirArbol(nuevaRaiz, viejosArboles)`, entonces:

1. x se refiere a un objeto recién creado;
2. `raiz(`x`) = nuevaRaiz`;
3. `subarboles(`x`) = viejosArboles`;

```
Object raiz(Arbol a)
```

Condición previa: ninguna.

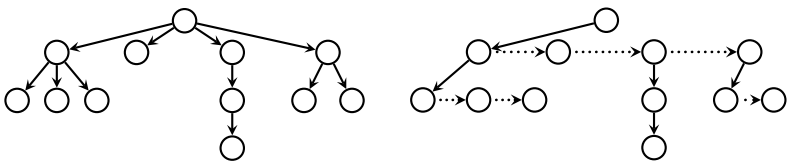
```
ListaArboles subarboles(Arbol a)
```

Condición previa: ninguna.

El TDA `ListaArboles` es el análogo de `ListaInt` con la clase `Arbol` en lugar de la clase **`int`** como tipo de los elementos. He aquí los prototipos.

```
ListaArboles cons(Arbol a, ListaArboles rHermanos)
Arbol primero(ListaArboles hermanos)
ListaArboles resto(ListaArboles hermanos)
ListaArboles nil
```

Figura 2.11 Especificaciones del TDA `Arbol` (general). Las especializaciones en las que los nodos pertenecen a una clase más específica que **`Object`** se definen de forma análoga.



(a) estructura lógica (b) estructura basada en listas

Figura 2.12 (a) Estructura lógica (o conceptual) de un árbol afuera general y (b) la representación correspondiente en la que los subárboles principales están en una lista: hacia abajo, las flechas continuas van a subárboles de extrema izquierda, y horizontalmente, las flechas punteadas van a subárboles hermanos derechos.

El primer subárbol principal, digamos t_0, es el *subárbol de extrema izquierda*; la raíz de t_0 es el *hijo de extrema izquierda*. Para cualquier subárbol principal, t_i, el siguiente subárbol principal, t_{i+1}, es el *subárbol hermano derecho* de t_i, si existe. La raíz de t_{i+1} es el *hermano derecho* de la raíz de t_i. En la figura 2.12 se da un ejemplo. A pesar de esta nomenclatura para la estructura de datos, reiteramos que el orden relativo de los subárboles de la lista se considera incidental para el

```
void recorrer(Arbol T)
     ListaArboles quedanSub;
     Procesar-orden previo Arbol.raiz(T);
     quedanSubarboles = Arbol.subarboles(T);
     while (quedanSubarboles ≠ ListaArboles.nil);
        Arbol subarbol = ListaArboles.primero(quedanSubarboles);
        recorrer(subarbol);
        Procesar-orden interno Arbol.raiz(T) y subarbol;
        quedanSubarboles = ListaArboles.resto(quedanSubarboles);
     Procesar-orden posterior Arbol.raiz(T);
     return;
```

Figura 2.13 El esqueleto de recorrido de un árbol general

árbol abstracto. El constructor del TDA, `construirArbol`, combina un nodo raíz y una lista de árboles para crear un árbol más grande.

Detalle de Java: Para definir estos TDA interrelacionados en Java con el control de visibilidad ideal, se recomienda usar el recurso de *paquete* de Java. Se requieren dos archivos en el mismo directorio, y deben llevar los nombres `Arbol.java` y `ListaArboles.java`. Los detalles no son difíciles, pero rebasan el alcance de este libro. Si se colocan los clientes y los TDA en el mismo directorio no habrá necesidad de manejar paquetes.

El recorrido de árboles se puede expresar con una extensión lógica del recorrido de árboles binarios (sección 2.3.3), con el esqueleto que se muestra en la figura 2.13. Los subárboles se recorren dentro de un ciclo **`while`** porque su número no está definido, y existe un número indeterminado de veces de orden interno. (Se incluyen aquí calificadores de nombre de clase, porque intervienen dos clases.) El tipo devuelto de `recorrer` variará dependiendo de la aplicación, y podrían requerirse parámetros adicionales. La figura 2.13 muestra el esqueleto común.

2.3.5 TDA de árbol adentro

Es un padre sabio el que conoce a su propio hijo.
—Shakespeare, *El mercader de Venecia*

Por lo regular, el patrón de acceso en un árbol va de la raíz hacia las hojas, y suele representarse en dirección descendente. Sin embargo, hay casos en los que es deseable (o suficiente) que el acceso esté orientado de las hojas hacia la raíz (hacia arriba, vea la figura 2.10b), y el acceso hacia abajo es innecesario. Un *árbol adentro* es un árbol que *únicamente* tiene este tipo de acceso: un nodo no "conoce" a sus hijos.

Un concepto importante en materia de árboles adentro es el de *antepasado*, que puede definirse recursivamente como sigue.

Definición 2.2

Un nodo v es un *antepasado* de sí mismo. Si p es el padre de v, entonces todo antepasado de p también es un *antepasado* de v. Lo opuesto a antepasado es *descendiente*. ■

En un árbol adentro, un nodo puede acceder a sus antepasados, pero no a sus descendientes. A diferencia del TDA `Arbol` común, en el que el objeto es un árbol entero, un objeto de un árbol adentro es un nodo y sus antepasados, por lo que la clase se llama `NodoArbolAdentro` y el constructor del TDA es `crearNodo`.

Sea v un objeto de la clase `NodoArbolAdentro`. ¿Qué funciones de acceso necesitamos para movernos en él? La primera función de acceso que se requiere es `esRaiz`(v), una función booleana que devuelve **`true`** si v no tiene padre. La segunda función de acceso es `padre`(v), que tiene como condición previa que `esRaiz`(v) sea **`false`**. En otras palabras, siempre que `esRaiz`(v) sea **`true`**, será un error invocar a `padre`(v).

La figura 2.14 contiene las especificaciones del TDA `NodoArbolAdentro`. Cuando se construye un nodo con `crearNodo`, es el único nodo de su árbol, así que `esRaiz` es **`true`**. Obviamente, necesitamos alguna forma de construir árboles más grandes. A diferencia de los TDA que vimos antes, éste usa un *procedimiento de manipulación* para tener funcionalidad. Recuerde que los procedimientos de manipulación siempre son de tipo **`void`**; no devuelven ningún valor. El procedimiento de manipulación es `hacerPadre`(v, p), que establece a p como padre de v. Su condición previa es que v no debe ser un antepasado de p (pues en tal caso se crearía un ciclo). Las condiciones posteriores son que `esRaiz`(v) es **`false`** y que `padre`(v) devuelve p.

Dependiendo de la aplicación, a menudo es necesario mantener algún tipo de datos en los nodos. Puesto que esto no afecta la estructura del árbol, podemos definir un par de operaciones sencillas, `ponerDatosNodo` y `datosNodo`, para que el cliente pueda almacenar y recuperar tales datos. Si bien los datos de los nodos podrían ser de diversos tipos, dependiendo de la aplicación, los definimos como **`int`** porque ése es el tipo más común. Por ejemplo, aunque un nodo no conoce a sus descendientes, es posible mantenerse al tanto de *cuántos* descendientes tiene cada nodo (ejercicio 2.12).

Los árboles adentro suelen estar incorporados a alguna otra estructura de datos, en lugar de ser TDA por derecho propio. Esto casi es necesario porque no existe un nodo desde el cual se pueda acceder a todo el árbol. Encontraremos árboles adentro en algoritmos para árboles abarcantes mínimos, caminos más cortos en grafos, y en la implementación del TDA Unión-Hallar. A su vez, el TDA Unión-Hallar se usa en diversos algoritmos, incluido uno para bosques abarcantes mínimos.

2.4 Pilas y colas

Las pilas y colas ilustran el siguiente nivel de complejidad en las especificaciones de tipos de datos abstractos. Sus TDA incluyen procedimientos de manipulación, para que los objetos de estas clases puedan cambiar de "estado". Ahora las especificaciones necesitan describir los cambios de estado que pueden presentarse. No obstante, todas las operaciones con estos versátiles TDA se pueden implementar en tiempo constante sin demasiada dificultad. Las pilas y colas son útiles para seguir la pista a tareas que deben efectuarse en situaciones en las que una tarea podría generar un número impredecible de tareas.

2.4.1 TDA de pila

Una *pila* es una estructura lineal en la que las inserciones y eliminaciones siempre se efectúan en un extremo, llamado *tope*. Esta política de actualización se denomina *último en entrar, primero en salir* (*LIFO*, *last input, first output,* por sus siglas en inglés). El elemento que está en el tope de la pila es el que se insertó más recientemente, y sólo puede inspeccionarse este elemento. Apilar

`NodoArbolAdentro crearNodo(int d)`

Condición previa: ninguna.

Condiciones posteriores: Si x = `crearNodo(d)`, entonces:

1. x se refiere a un objeto recién creado;
2. `datosNodo(`x`)` = d;
3. `esRaiz(`x`)` = **`true`**;

`boolean esRaiz(NodoArbolAdentro v)`

Condición previa: ninguna.

`NodoArbolAdentro padre(NodoArbolAdentro v)`

Condición previa: `esRaiz(`v`)` = **`false`**.

`int datosNodo(NodoArbolAdentro v)`

Condición previa: ninguna.

`void hacerPadre(NodoArbolAdentro v, NodoArbolAdentro p)`

Condición previa: el nodo v no es antepasado de p.

Condiciones posteriores:

1. `datosNodo(`v`)` no cambia;
2. `padre(`v`)` = p;
3. `esRaiz(`v`)` = **`false`**;

`void ponerDatosNodo(NodoArbolAdentro v, int d)`

Condición previa: ninguna.

Condiciones posteriores:

1. `datosNodo(`v`)` = d;
2. `padre(`v`)` no cambia;
3. `esRaiz(`v`)` no cambia;

Figura 2.14 Especificaciones del TDA `NodoArbolAdentro`. La función `crearNodo` es el constructor; `esRaiz`, `padre` y `datosNodo` son funciones de acceso; `hacerPadre` y `ponerDatosNodo` son procedimientos de manipulación. Las especializaciones en las que los datos de los nodos pertenecen a una clase distinta de **`int`** se definen de forma análoga.

(`push`) un elemento significa insertarlo en la pila. Desapilar (`pop`) significa eliminar el elemento que está en el tope. Podemos acceder al elemento que está en el tope de una pila no vacía con `top`. La práctica moderna dicta no combinar las funciones de `top` y `pop` en una sola operación. La figura 2.15 presenta las especificaciones del TDA `Pila`.

A diferencia de las especificaciones de TDA anteriores, no es posible decir explícitamente qué valores devolverán las funciones de acceso `estaVacia` y `top` después de un `pop`. Por tanto, se requiere una sección de *Explicación* para dar información acerca de las *sucesiones* de opera-

`Pila crear()`

Condición previa: ninguna.

Condiciones posteriores: Si *s* = `crear()`, entonces:

1. *s* se refiere a un objeto recién creado;
2. `estaVacia`(*s*) = **`true`**;

`boolean` `estaVacia(Pila` *s*`)`

Condición previa: ninguna.

`Object` `top(Pila` *s*`)`

Condición previa: `estaVacia`(*s*) = **`false`**.

`void` `push(Pila` *s*`,` **`Object`** *e*`)`

Condición previa: ninguna.

Condiciones posteriores:

1. `top`(*s*) = *e*;
2. `estaVacia`(*s*) = **`false`**;

`void` `pop(Pila` *s*`)`

Condición previa: `estaVacia`(*s*) = **`false`**.

Condiciones posteriores: véase la explicación que sigue.

Explicación: Después de `crear`, cualquier sucesión válida de operaciones `push` y `pop` (es decir, nunca se desapila más de lo que se apila, acumulativamente) produce el mismo estado de pila que cierta sucesión que consiste únicamente en operaciones `push`. Para obtener esta sucesión, se buscan de forma repetitiva todos los `pop` que vayan precedidos inmediatamente por un `push` y se elimina ese par de operaciones de la sucesión. En esto se aprovecha el *Axioma de Pila*: un `push` seguido de un `pop` no tiene ningún efecto neto sobre la pila.

Figura 2.15 Especificaciones del TDA `Pila`. El constructor es `crear`; `estaVacia` y `top` son funciones de acceso; `push` y `pop` son procedimientos de manipulación. Las especializaciones en las que los elementos pertenecen a una clase más específica que **`Object`** se definen de forma análoga.

ciones de apilar y desapilar. En el ejercicio 2.13 se da un ejemplo. La sección de Explicación describe de forma indirecta las condiciones posteriores de `pop`. La técnica de especificar propiedades, o invariantes, de sucesiones de operaciones permite especificar lógicamente el TDA sin hacer mención de aspectos de la implementación a los que el cliente no tiene acceso. Esta técnica se necesita a menudo en el caso de TDA más complejos.

En casi todos los casos en que se necesitaría la pila como estructura explícita, ésta se hace innecesaria si se usan procedimientos recursivos, pues el sistema "de tiempo de ejecución" implementa una pila de variables locales para cada invocación de función. Se puede implementar una

pila en un arreglo o con base en el TDA de lista. De cualquier manera, todas las operaciones se podrán implementar en tiempo $\Theta(1)$. Si no se conoce con antelación el tamaño máximo que podría alcanzar la pila, se puede utilizar una técnica de duplicación de arreglo para expandir su tamaño (véase la sección 6.2). Este detalle de la implementación puede ocultarse a los clientes del TDA `Pila`.

2.4.2 TDA de cola

Una *cola* es una estructura lineal en la que todas las inserciones se efectúan por un extremo, la parte de *atrás*, y todas las eliminaciones se efectúan en el otro extremo, el *frente*. Sólo puede inspeccionarse el elemento frontal. Esta política de actualización se denomina *primero que entra, primero que sale* (*FIFO*, *first input, first output,* por sus siglas en inglés). Los procedimientos de manipulación son *encolar* para insertar y *desencolar* para eliminar. Tenemos las funciones de acceso `estaVacia` y `frente` para probar si la cola está vacía y, si no lo está, acceder al elemento frontal. La figura 2.16 presenta las especificaciones del TDA `Cola`.

Al igual que con el TDA `Pila`, no es posible decir explícitamente qué valores devolverán las funciones de acceso después de un `desencolar`. Por ello, se requiere una sección de *Explicación* para dar información acerca de las *sucesiones* de operaciones de encolar y desencolar. En el ejercicio 2.13 se presenta un ejemplo.

Las colas se pueden implementar de forma eficiente (todas las operaciones en $\Theta(1)$) empleando un arreglo. Si no se conoce con antelación el tamaño máximo que podría alcanzar la cola, se puede emplear una técnica de duplicación de arreglos para expandirla (véase la sección 6.2). Este detalle de la implementación se puede ocultar a los clientes del TDA `Cola`. También, se puede definir una variante actualizable del TDA de lista, en el que se pueda actualizar el valor de `resto` para anexar una lista al extremo trasero de una lista existente (véase el apéndice A). Entonces, `encolar` anexará una lista que consiste únicamente en el elemento nuevo al final de la cola actual. Para que esta operación esté en $\Theta(1)$ es preciso mantener una referencia al último elemento de la cola, además de una referencia a toda la cola.

2.5 TDA para conjuntos dinámicos

Un *conjunto dinámico* es un conjunto cuyos elementos cambian mientras se ejecuta el algoritmo que usa el conjunto. En muchos casos, el objetivo del algoritmo es construir el conjunto mismo, pero para ello necesita acceder al conjunto conforme se va construyendo para determinar cómo seguir con la construcción. El conjunto apropiado de operaciones para un TDA de conjunto dinámico varía ampliamente, dependiendo de las necesidades del algoritmo o la aplicación que lo esté usando. Entre los ejemplos estándar están las colas de prioridad, las colecciones de conjuntos disjuntos que requieren operaciones de unión y hallar, y los diccionarios. Describiremos dichos ejemplos en esta sección.

Los conjuntos dinámicos son los que imponen restricciones más estrictas a sus estructuras de datos. Para ninguno de los TDA de esta sección es posible implementar todas las operaciones requeridas en tiempo constante. Es preciso hacer concesiones y diferentes implementaciones que resultarán más eficientes para diferentes aplicaciones. La búsqueda de eficiencia ha dado pie a varias implementaciones en extremo avanzadas y complejas, algunas de las cuales se mencionan en el capítulo 6.

`Cola crear()`

Condición previa: ninguna.

Condiciones posteriores: Si q = `crear()`, entonces:

1. q se refiere a un objeto recién creado;
2. `estaVacia`(q) = **`true`**;

`boolean` `estaVacia(Cola` q`)`

Condición previa: ninguna.

`Object` `frente(Cola` q`)`

Condición previa: `estaVacia`(q) = **`false`**.

`void` `encolar(Cola` q`,` **`Object`** e`)`

Condición previa: ninguna.

Condiciones posteriores: denotemos con /q/ el estado de q antes de la operación.

1. Si `estaVacia`(/q/) = **`true`**, `frente`(q) = e,
2. Si `estaVacia`(/q/) = **`false`**, `frente`(q) = `frente`(/q/).
3. `estaVacia`(q) = **`false`**;

`void` `desencolar(Cola` q`)`

Condición previa: `estaVacia`(q) = **`false`**.

Condiciones posteriores: véase la explicación que sigue.

Explicación: Después de `crear`, cualquier sucesión válida de operaciones `encolar` y `desencolar` (es decir, nunca se desencola más de lo que se encola, acumulativamente) produce el mismo estado de cola que cierta sucesión que consiste únicamente en operaciones `encolar`. Para obtener esta sucesión, se buscan de forma repetitiva el primer (más antiguo) `desencolar` y el primer `encolar` y se elimina ese par de operaciones de la sucesión. Las funciones de acceso `frente`(q) y `estaVacia`(q) adoptan los mismos valores que tendrían después de esta sucesión equivalente, que consiste exclusivamente en operaciones `encolar`.

Figura 2.16 Especificaciones del TDA `Cola`. El constructor es `crear`; `estaVacia` y `frente` son funciones de acceso; `encolar` y `desencolar` son procedimientos de manipulación. Las especializaciones en las que los elementos pertenecen a una clase más específica que **`Object`** se definen de forma análoga.

2.5.1 TDA de cola de prioridad

Una *cola de prioridad* es una estructura que posee algunos aspectos de las colas FIFO (sección 2.4.2) pero en la que el orden de los elementos está relacionado con la *prioridad* de cada elemento, no con el momento cronológico en que llegó. La prioridad de los elementos (también llamada "clave") es un parámetro que se proporciona a la operación `insertar`, no alguna propiedad innata que el TDA conoce. Supondremos que su tipo es **`float`**, para ser más específicos. También supondremos que los elementos son de tipo **`int`** porque éste es el tipo que se usa en la mayor parte de las aplicaciones de optimización. En la práctica, los elementos tienen un *identificador* que es un **`int`**, además de otros campos de datos asociados; este identificador no debe confundirse con la "clave", que es el nombre tradicional del campo de prioridad.

Conforme se inserta cada elemento en una cola de este tipo, la inserción se efectúa *conceptualmente* en orden según su prioridad. El único elemento que se puede inspeccionar y sacar es el elemento *más importante* que está actualmente en la cola de prioridad. En realidad, lo que ocurre tras bambalinas depende de la implementación, en tanto la *apariencia* externa sea en todo aspecto congruente con esta vista.

La idea de prioridad puede ser que el elemento más importante tiene la prioridad más baja (una perspectiva de costo) o bien que tiene la prioridad más alta (una perspectiva de utilidades). En problemas de optimización prevalece la perspectiva de costo, y es por ello que los nombres históricos de ciertas operaciones de cola de prioridad reflejan este punto de vista: `obtenerMin`, `borrarMin` y `decrementarClave`.

Una aplicación importante de la cola de prioridad es el método de ordenamiento conocido como *Heapsort* (sección 4.8), cuyo nombre proviene de la implementación con *montón* (*heap*, en inglés) de la cola de prioridad. En el caso de *Heapsort*, la clave *más grande* se considera la más importante, por lo que los nombres apropiados son `obtenerMax` y `borrarMax` en este contexto.

A diferencia de las colas FIFO, las colas de prioridad no se pueden implementar de forma tal que todas las operaciones estén en $\Theta(1)$. Es preciso considerar concesiones entre métodos de implementación opuestos, junto con las necesidades de un algoritmo o aplicación en particular, para llegar a una opción que ofrezca la más alta eficiencia en general. Estas cuestiones se estudiarán junto con los diversos algoritmos que usan colas de prioridad. Además de *Heapsort* (sección 4.8), existe una familia de algoritmos llamados *algoritmos codiciosos*, que suelen usar una cola de prioridad, e incluyen los algoritmos de árboles abarcantes mínimos de Prim y de Kruskal (secciones 8.2 y 8.4), el algoritmo de camino más corto de origen único de Dijkstra (sección 8.3) y ciertos algoritmos de aproximación para problemas *NP*-difíciles (capítulo 13). El *método codicioso* es un importante paradigma del diseño de algoritmos.

Pasemos ahora a las especificaciones del TDA de cola de prioridad, que se muestran en las figuras 2.17 y 2.18. Son evidentes similitudes con el TDA de cola (FIFO). Una divergencia importante es que la operación de eliminar es `borrarMin` que, como su nombre implica, elimina el elemento con el campo de prioridad más bajo ("clave" mínima), no el elemento más antiguo.

Otro cambio importante es que el orden de prioridad se puede reacomodar con la operación `decrementarClave`. Sin embargo, esta operación y la función `obtenerPrioridad` se pueden omitir de las implementaciones destinadas a *Heapsort* y otras aplicaciones que no necesitan estas capacidades; añaden complicaciones considerables tanto a la especificación como a la implementación (como veremos en la sección 6.7.1). Usamos el término "cola de prioridad *elemental*" para referirnos al TDA que no tiene `decrementarClave` ni `obtenerPrioridad`.

`ColaPrioridad crear()`

Condición previa: ninguna.

Condiciones posteriores: Si `pq` = `crear()`, entonces `pq` se refiere a un objeto recién creado y `estaVacia(pq)` = **`true`**.

`boolean` `estaVacia(ColaPrioridad pq)`

Condición previa: ninguna.

`int` `obtenerMin(ColaPrioridad pq)`

Condición previa: `estaVacia(pq)` = **`false`**.

`void` `insertar(ColaPrioridad pq,` **`int`** `id,` **`float`** w)

Condición previa: Si se implementa `decrementarClave` (véase la figura 2.18), entonces `id` no debe estar ya en `pq`.

Condiciones posteriores: el identificador del elemento a insertar es `id` y la prioridad es w.

1. `estaVacia(pq)` = **`false`**;
2. Si se implementa `obtenerPrioridad` (véase la figura 2.18), entonces `obtenerPrioridad(pq, id)` = w.
3. Vea la explicación, más abajo, en lo referente al valor de `obtenerMin(pq)`.

`void` `borrarMin(ColaPrioridad pq)`

Condición previa: `estaVacia(pq)` = **`false`**.

Condiciones posteriores:

1. Si el número de operaciones `borrarMin` es menor que el de operaciones `insertar` desde `crear(pq)`, entonces `estaVacia(pq)` = **`false`**, de lo contrario, es **`true`**.
2. Vea la explicación que sigue en lo tocante al valor de `obtenerMin(pq)`.

Explicación: Pensemos en `/pq/` (el estado de `pq` antes de la operación en cuestión) de manera abstracta como una sucesión de pares $((\texttt{id}_1, w_1), (\texttt{id}_2, w_2), \ldots, (\texttt{id}_k, w_k))$, en orden no decreciente según los valores de w_i, que representan las prioridades de los elementos $\texttt{id}_i$. Entonces, `insertar`(`pq`, `id`, w) inserta (`id`, w) en esta sucesión en orden, extendiendo `pq` a $k + 1$ elementos en total. Además, `borrarMin(pq)` elimina el primer elemento de la sucesión `/pq/`, dejando a `pq` con $k - 1$ elementos. Por último, `obtenerMin(pq)` devuelve $\texttt{id}_1$.

Figura 2.17 Especificaciones del TDA de cola de prioridad *elemental* (`ColaPrioridad`). El constructor es `crear`; `estaVacia` y `obtenerMin` son funciones de acceso; `insertar` y `borrarMin` son procedimientos de manipulación. En la figura 2.18 se especifican operaciones adicionales para un TDA de cola de prioridad *completa*. Las especializaciones en las que los elementos pertenecen a una clase distinta de **`int`** se definen de forma análoga.

```
float obtenerPrioridad(ColaPrioridad pq, int id)
```

Condición previa: `id` está "en" `pq`.

```
void decrementarClave(ColaPrioridad pq, int id, float w)
```

Condición previa: `id` está "en" `pq` y $w <$ `obtenerPrioridad(pq, id)`. Es decir, la nueva prioridad w debe ser menor que la prioridad actual de ese elemento.

Condiciones posteriores: `estaVacia(pq)` sigue siendo **`false`**. `obtenerPrioridad(pq, id)` $= w$. Véase la explicación que sigue en lo tocante al valor de `obtenerMin(pq)`.

Explicación: Al igual que en la explicación de la figura 2.17, pensemos en `/pq/` de manera abstracta como una sucesión de pares $((\texttt{id}_1, w_1), (\texttt{id}_2, w_2), \ldots, (\texttt{id}_k, w_k))$, en orden según los valores de w_i. Además, todos los `id` son únicos. Entonces, `decrementarClave(pq, id, w)` requiere que `id` $= \texttt{id}_i$ para alguna $1 \leq i \leq k$, y de hecho elimina $(\texttt{id}_i, w_i)$ de la sucesión `/pq/`, y luego inserta $(\texttt{id}_i, w)$ en la sucesión en orden según w. La sucesión final sigue teniendo k elementos. Igual que antes, `obtenerMin(pq)` devuelve $\texttt{id}_1$.

Figura 2.18 Especificaciones de operaciones adicionales que sólo se definen para un TDA de cola de prioridad *completa* (`ColaPrioridad`). En la figura 2.17 se presentan todas las demás operaciones. Aquí, `obtenerPrioridad` es una función de acceso, y `decrementarClave` es un procedimiento de manipulación.

2.5.2 TDA de Unión-Hallar para conjuntos disjuntos

El TDA *Unión-Hallar* se llama así por sus dos principales operaciones, pero también se conoce como TDA de *Conjuntos Disjuntos*. En un principio, todos los elementos de interés se colocan en conjuntos individuales de un solo elemento con la operación de constructor `crear` o bien se añaden individualmente con el procedimiento de manipulación `hacerConjunto`. La función de acceso `hallar` devuelve el *identificador de conjunto* actual de un elemento. Mediante una operación de `union` podemos combinar dos conjuntos, después de lo cual dejarán de existir como entidades individuales. Por tanto, ningún elemento puede estar en más de un conjunto. En la práctica, es común que los elementos sean enteros y el identificador de conjunto sea algún elemento específico del conjunto, llamado *líder*. No obstante, en teoría los elementos pueden ser de cualquier tipo y los identificadores de conjunto no tienen que ser del mismo tipo que los elementos.

No hay forma de "hacer un recorrido" por todos los elementos de un conjunto. Observe la similitud con los árboles adentro (sección 2.3.5), en los que no hay forma de recorrer todo el árbol. De hecho, los árboles adentro pueden servir para implementar de forma eficaz el TDA Unión-Hallar. En la sección 6.6 se describen de forma detallada las implementaciones de este TDA. Las especificaciones de `UnionHallar` se presentan en la figura 2.19.

2.5.3 TDA de diccionario

Un diccionario es una estructura de almacenamiento asociativo general. Es decir, los elementos tienen un identificador de algún tipo y contienen cierta información que es preciso almacenar y recuperar. La información está *asociada* al identificador. El nombre de "diccionario" para este TDA proviene de la analogía con los diccionarios comunes, en los que las palabras son sus propios identificadores, y las definiciones, pronunciaciones y demás son la información asociada. Sin

```
UnionHallar crear(int n)
```

Condición previa: ninguna.

Condiciones posteriores: Si `conjuntos` = `crear`(n), entonces `conjuntos` se refiere a un objeto recién creado; `hallar`(`conjuntos`, e) = e para $1 \leq e \leq n$, y no está definido para otros valores de e.

```
int hallar(UnionHallar conjuntos, e)
```

Condición previa: El conjunto $\{e\}$ ya se creó, sea con `hacerConjunto`(`conjuntos`, e) o con `create`.

```
void hacerConjunto(UnionHallar conjuntos, int e)
```

Condición previa: `hallar`(`conjuntos`, e) no está definido.

Condiciones posteriores: `hallar`(`conjuntos`, e) = e; es decir, e es el identificador de un conjunto de un solo elemento que contiene e.

```
void union(UnionHallar conjuntos, int s, int t)
```

Condiciones previas: `hallar`(`conjuntos`, s) = s y `hallar`(`conjuntos`, t) = t, es decir, tanto s como t son identificadores de conjunto, o "líderes". Además, $s \neq t$.

Condiciones posteriores: Denotemos con `/conjuntos/` el estado de `conjuntos` antes de la operación. Entonces, para toda x tal que `hallar`(`/conjuntos/`, x) = s, o `hallar`(`/conjuntos/`, x) = t, ahora tenemos `hallar`(`conjuntos`, x) = u. El valor de u será s o bien t. Todas las demás invocaciones de `hallar` devuelven el mismo valor que devolvían antes de la operación de unión.

Figura 2.19 Especificaciones del TDA `UnionHallar`. El constructor es `crear`; `hallar` es una función de acceso; `hacerConjunto` y `union` son procedimientos de manipulación.

embargo, la analogía no debe llevarse demasiado lejos, porque en un TDA de diccionario los identificadores no tienen un orden implícito. El aspecto importante de un diccionario es que en cualquier momento se puede recuperar cualquier información almacenada en él.

Las especificaciones de Diccionario se presentan en la figura 2.20. Dichas especificaciones serán "preliminares" en tanto no se especifique el tipo (o clase) de `IdDicc`. Éste es el tipo o clase del *identificador* para los elementos del diccionario. Por lo regular será la clase integrada **`String`**, el tipo primitivo **`int`**, o una clase organizadora que agrupe varios de estos tipos. Una de las ventajas de diseñar con el TDA Diccionario es que esta decisión se puede posponer hasta que se haya diseñado el algoritmo que usa este Diccionario. Podemos `crear` un diccionario vacío y luego `almacenar` pares `(id, info)` en él. Podemos averiguar si cualquier `id` es `miembro` del Diccionario, y en caso afirmativo, `recuperar` la información asociada. Para las aplicaciones de este libro no es necesario poder borrar elementos, pero en otras aplicaciones podría ser apropiada una operación `borrar`.

El TDA Diccionario es muy útil en el diseño de algoritmos de programación dinámica (capítulo 10). Los diccionarios también son prácticos para registrar nombres externos (por lo regular cadenas que se leen de la entrada) de modo que un programa pueda determinar si ya vio antes un nombre o lo está viendo por primera vez. Por ejemplo, los compiladores necesitan mantenerse al tanto de los nombres de datos y de procedimientos que ya se han usado.

`Dicc crear()`

Condición previa: ninguna.

Condiciones posteriores: Si d = `crear()`, entonces:

1. d se refiere a un objeto recién creado;
2. `miembro(`d`, id)` = **`false`** para todo `id`.

`boolean` `miembro(Dicc` d`, IdDicc id)`

Condición previa: ninguna.

`Object` `recuperar(Dicc` d`, IdDicc id)`

Condición previa: `miembro(`d`)` = **`true`**.

`void` `almacenar(Dicc` d`, IdDicc id,` **`Object`** `info)`

Condición previa: ninguna.
Condiciones posteriores:

1. `recuperar(`d`, id)` = `info`;
2. `miembro(`d`, id)` = **`true`**;

Figura 2.20 Especificaciones del TDA `Dicc`, que serán preliminares en tanto no se convierta `IdDicc` en un tipo o clase existente. El constructor es `crear`; `miembro` y `recuperar` son funciones de acceso; `almacenar` es un procedimiento de manipulación. Las especializaciones en las que los datos informativos pertenecen a una clase más específica que **`Object`** se definen de forma análoga.

Ejercicios

Sección 2.2 Especificación de TDA y técnicas de diseño

2.1 Considere algunas operaciones de TDA con las rúbricas de tipo que se muestran en seguida. El nombre de clase del TDA es `Gorp`. Con base en la definición 2.1, ¿a qué categoría podría pertenecer cada operación? Dé una explicación corta de sus respuestas.

a. **`void`** `warp(Gorp g)`.
b. `Gorp harp()`.
c. **`int`** `pork(Gorp g)`.
d. **`void`** `work(Gorp g,` **`int`** `i)`.
e. **`int`** `perk(Gorp g1, Gorp g2)`.
f. `Gorp park(Gorp g1, Gorp g2, Gorp g3,` **`int`** `i)`.

2.2 Usted necesita escribir el código que usa el tipo de datos abstracto `Gorp`, el cual otras personas ya han estado usando durante algún tiempo, aunque usted nunca se había topado con él. Además de la documentación con que cuenta, usted puede examinar `Gorp.java`, el archivo fuente del TDA, y `ProbadorGorp.java` que es un programa que usa el TDA. ¿Cuál archivo es pre-

ferible como fuente de información acerca de la mejor forma de usar las operaciones del TDA `Gorp`, y por qué?

Sección 2.3 TDA elementales: listas y árboles

2.3 Use las operaciones del TDA `ListaInt` (vea las especificaciones en la figura 2.3) para implementar las siguientes utilerías de listas, como *cliente* del TDA. Es decir, sus procedimientos están afuera de la clase `ListaInt`, por lo que no saben cómo están implementadas las listas. Este ejercicio es un buen calentamiento para las manipulaciones de listas que necesitarán los algoritmos de capítulos posteriores.

En el caso de tareas que implican recorrer una lista, trate de desarrollar un esqueleto común con variaciones para las distintas tareas, en lugar de enfocar cada tarea de diferente forma. Es aceptable, e incluso preferible, usar pseudocódigo claro en lugar de una sintaxis estricta, pero debe quedar claro cómo se usan las operaciones del TDA y qué valor devuelve el procedimiento, en su caso.

Si su procedimiento no funciona con todas las listas de enteros, no olvide plantear las condiciones previas que se deben cumplir para que funcione correctamente. (No se preocupe por desbordamientos del tamaño del tipo **`int`**.) No olvide las listas vacías.

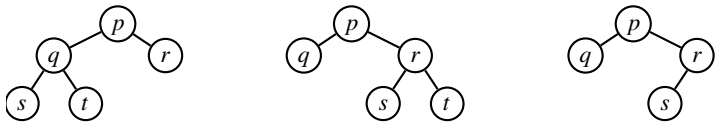
a. Contar el número de elementos de una lista (longitud de lista).
b. Sumar los elementos de una lista.
c. Multiplicar los elementos de una lista.
d. Devolver el elemento máximo de una lista.
e. Devolver el elemento mínimo de una lista.
f. Devolver una nueva lista que contenga los elementos de la lista original pero en el orden inverso.
g. Construir (y devolver) una lista de enteros leídos de la "entrada". Para no tener que preocuparse por lo que significa exactamente "entrada", suponga que cuenta con métodos llamados `masDatos` y `leerInt`, y que la entrada no contiene otra cosa que no sea números enteros. La función `masDatos` es **`boolean`** y devuelve **`true`** si y sólo si hay otro entero que leer. La función `leerInt` devuelve un entero que leyó de "entrada" y tiene como condición previa que `masDatos` devuelva **`true`**. Una vez que `leerInt` devuelve un entero, ese entero ya no está en la "entrada".
h. Distribuir los enteros de una lista según su magnitud, creando un arreglo de listas llamado `cubeta`. El arreglo `cubeta` tiene 10 elementos. Los elementos de la lista original que estén dentro del intervalo de 0 a 99 deberán colocarse en la lista `cubeta[0]`, los elementos que estén dentro del intervalo de 100 a 199 deberán colocarse en la lista `cubeta[1]`, y así sucesivamente, y todos los elementos que sean 900 o mayores se deberán colocar en la lista `cubeta[9]`. Suponga que su procedimiento recibe dos parámetros, la lista de elementos a distribuir y el arreglo `cubeta` (de modo que su procedimiento no necesita crear el arreglo, sino sólo inicializarlo y llenarlo).

2.4 Demuestre el lema 2.1.

2.5 Demuestre el lema 2.2.

2.6 Demuestre el lema 2.3.

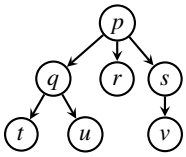
2.7 Dé una sucesión de operaciones del TDA `ArbolBin` para construir cada uno de los árboles binarios que se muestran. Declare una variable distinta para cada nodo, que será el subárbol cuya raíz es ese nodo. El valor de la raíz (devuelto por la función de acceso `raiz`) será el nombre del nodo, de tipo **`String`**. Por ejemplo, la variable llamada `q` almacena el subárbol cuya raíz es `q` en el diagrama, y `raiz(q) == "q"` una vez que se haya construido `q`.



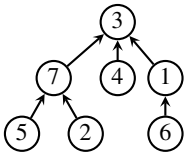
2.8 Implemente el TDA `Lista` (con elementos de tipo **`Object`**) utilizando las operaciones del TDA `ArbolBin` (vea las figuras 2.3 y 2.8). Es decir, trate la clase `Lista` como cliente de la clase `ArbolBin`.

***2.9** Implemente el TDA `ArbolBin` utilizando las operaciones del TDA `Lista` (con elementos de tipo **`Object`**) (vea las figuras 2.3 y 2.8). Es decir, trate la clase `ArbolBin` como cliente de la clase `Lista`.

2.10 Dé una sucesión de operaciones de `Arbol` y/o `ListaArboles` para construir el árbol (afuera) general que se muestra en seguida. Declare una variable distinta para cada nodo, que contendrá el subárbol cuya raíz es ese nodo. El valor de la raíz será el nombre del nodo, de tipo **`String`**. Por ejemplo, la variable llamada `s` almacena el subárbol cuya raíz es `s` en el diagrama, y el valor de la raíz de ese subárbol es `"s"`.



2.11 Dé una sucesión de operaciones del TDA `NodoArbolAdentro` para construir el árbol adentro que se muestra en seguida. Suponga que los nodos están en un arreglo llamado `nodoAdentro` y que `datosNodo` de cada nodo es su propio índice dentro de dicho arreglo. El índice de un nodo es el número que se muestra en el diagrama. Por ejemplo, `nodoAdentro[3]` contiene el nodo raíz.



2.12 Escriba los procedimientos de biblioteca `crearNodoDimensionado` y `hacerPadreDimensionado` para saber cuántos nodos hay en cada subárbol de un árbol adentro y también efec-

tuar las operaciones que efectúan `crearNodo` y `hacerPadre`. Al interactuar con las operaciones del TDA `NodoArbolAdentro` de la figura 2.14, sus procedimientos deberán hacer que `datosNodo`(v) devuelva el número de nodos que tiene el subárbol cuya raíz es v. La función `crearNodoDimensionado` no deberá recibir parámetros y deberá devolver un `NodoArbolAdentro`. (¿Qué valor deberá devolver `datosNodo` para el nodo devuelto por `crearNodoDimensionado`?) El procedimiento `hacerPadreDimensionado` tiene la misma rúbrica de tipo que `hacerPadre`. *Sugerencia:* Piense detenidamente en qué dimensiones del árbol cambian y qué tanto cambian como resultado de una operación `hacerPadre`. Se deben considerar varios casos.

Sección 2.4 Pilas y colas

2.13 Considere la sucesión de operaciones: agregar(1), agregar(2), borrar, agregar(3), agregar(4), agregar(5), agregar(6), borrar, borrar, agregar(7), borrar, borrar.

a. Interpretando agregar y borrar como operaciones de Pila, `push` y `pop`, dé una sucesión equivalente sin operaciones borrar. ¿Qué devolvería `top` después de ejecutarse esa sucesión?
b. Repita, pero interpretando agregar y borrar como operaciones de Cola, `encolar` y `desencolar`. ¿Qué devolvería `frente` después de ejecutarse la sucesión?

2.14 Bosqueje una implementación del TDA Pila, según las especificaciones de la figura 2.15, empleando el TDA Lista, según las especificaciones de la figura 2.3; es decir, trate la clase `Pila` como cliente de la clase `Lista`. Suponiendo que cada operación de Lista se ejecuta en $O(1)$ (tiempo constante), ¿cuánto tarda cada operación de Pila con su implementación?

2.15 Considere una variante del TDA Pila en la que el constructor tiene un parámetro entero n cuyo significado es que la pila nunca debe contener más de n elementos; es decir, la rúbrica es `Pila crear(int, n)`. Sin embargo, debido a combinaciones de `push` y `pop`, podrían ejecutarse muchas más de n operaciones `push` durante la vida de la pila.

a. ¿Cómo deberán modificarse las especificaciones para tener en cuenta este nuevo parámetro? Evite cambios drásticos. *Sugerencia:* Considere las condiciones previas.
b. Bosqueje una implementación basada en almacenar los elementos de la pila en un arreglo, que se construye con `crear`. (Todas las operaciones se pueden ejecutar en tiempo constante con una buena implementación.)
c. Ahora considere la restricción aún más drástica de que no se efectúen más de n operaciones `push` durante la vida de la pila. ¿Puede simplificar su implementación? Explique.

2.16 Considere una variante del TDA Cola en la que el constructor tiene un parámetro entero n cuyo significado es que la cola nunca debe contener más de n elementos; es decir, la rúbrica es `Cola crear(int, n)`. Sin embargo, debido a combinaciones de `encolar` y `desencolar`, podrían ejecutarse muchas más de n operaciones `encolar` durante la vida de la cola.

a. ¿Cómo deberán modificarse las especificaciones para tener en cuenta este nuevo parámetro? Evite cambios drásticos. *Sugerencia:* Considere las condiciones previas.
b. Bosqueje una implementación basada en almacenar los elementos de la cola en un arreglo, que se construye con `crear`. No olvide considerar dónde `encolar` coloca los elementos nuevos, sobre todo cuando se efectúan más de n operaciones `encolar`. ¿Cómo funcionan `fren-`

te y desencolar?, ¿cómo se detecta una cola vacía?, ¿se puede distinguir una cola vacía de una que contiene n elementos? (Todas las operaciones se pueden ejecutar en tiempo constante con una buena implementación.)

c. Ahora considere la restricción aún más drástica de que no se efectuarán más de n operaciones encolar en total durante la vida de la cola. (Después encontraremos algoritmos para los cuales ésta es una restricción práctica.) ¿Puede simplificar su implementación? Explique.

Sección 2.5 TDA para conjuntos dinámicos

2.17 Para cada parte de este ejercicio, bosqueje una implementación sencilla del TDA de cola de prioridad según las especificaciones de la figura 2.17, empleando el TDA Lista especificado en la figura 2.3; es decir, trate la clase ColaPrioridad como cliente de la clase Lista. Su clase podría incluir algunos otros campos de ejemplar, pero la Lista deberá ser la estructura de datos principal para almacenar los elementos de la cola de prioridad. Describa las ideas principales; no es necesario escribir el código.

a. Logre que la operación insertar se ejecute en $O(1)$ (tiempo constante). ¿Cuánto tardan en ejecutarse las demás operaciones, en el peor caso, si la cola de prioridad contiene n elementos?

b. Logre que la operación borrarMin se ejecute en $O(1)$. ¿Cuánto tardan en ejecutarse las demás operaciones, en el peor caso, si la cola de prioridad contiene n elementos?

c. Suponga que puede usar un arreglo en lugar de una lista ligada para almacenar los elementos. (No se preocupe por desbordamientos: suponga que de alguna manera se puede hacer lo bastante largo.) *Utilizando las mismas ideas generales* que usó en las partes (a) y (b), ¿alguna de las operaciones tendrá un mejor orden asintótico para su tiempo de ejecución? Explique. (En capítulos posteriores veremos algunas implementaciones avanzadas de colas de prioridad, y necesitarán arreglos.)

Problemas adicionales

***2.18** Usted tiene un conjunto de nodos de árbol adentro almacenados en un arreglo llamado nodoAdentro, en las posiciones $1, \ldots, n$. El valor de datosNodo para cada nodo es el índice del nodo dentro del arreglo nodoAdentro. En otras palabras, para $1 \leq v \leq n$, datosNodo(nodoAdentro[v]) = v. Puede suponer que los nodos realmente forman un árbol adentro; es decir, esRaiz es **true** para exactamente un nodo, el padre de todos los demás nodos está en el mismo arreglo, y no hay ciclos que incluyan sucesiones de padres.

Diseñe un algoritmo para construir el árbol afuera correspondiente, utilizando los TDA NodoArbolAdentro y Arbol como cliente; su algoritmo no sabrá cómo están implementados esos TDA. Idealmente, su algoritmo se ejecutará en tiempo lineal, $\Theta(n)$. Podría ser útil emplear unos cuantos arreglos de trabajo y también podría ser útil un objeto Pila.

Sugerencia: Puesto que el TDA Arbol no tiene procedimientos de manipulación, el árbol afuera se deberá construir desde las hojas hacia la raíz. El bosquejo que sigue usa una técnica general llamada *podado de origen* o nombres similares.

Inicialice un arreglo de contadores, llamado restantes, para registrar el número de hijos que tiene cada nodo para los cuales no se ha creado todavía su objeto Arbol. Inicialice otro arreglo de tipo ListaArboles, llamado subarboles, con listas vacías. Si el contador restantes

de un nodo es 0, se convertirá en un *origen* y se podrá crear su objeto `Arbol`. Una pila es un buen mecanismo para llevar el control de los orígenes. Cuando se crea un objeto `Arbol` para el nodo v (donde v es el índice que está en el `nodoAdentro`, un entero), se le puede insertar en la lista `subarboles` del padre de v, y el contador `restantes` de ese padre se puede decrementar en 1.

Notas y referencias

Los fundamentos de la especificación y diseño de tipos de datos abstractos que presentamos en este capítulo se deben a Parnas (1972). Parnas fue uno de los primeros investigadores en hacer hincapié en la necesidad de encapsular los datos, este hecho tuvo una influencia notable sobre el desarrollo de la programación orientada a objetos (OOP, por sus siglas en inglés).

Hay numerosos textos sobre estructuras de datos que podrían servir para repasar y como referencia; por ejemplo, Roberts (1995), Kruse, Tondo y Leung (1997) y Weiss (1998).

3

Recursión e inducción

3.1 Introducción

El profesor John McCarthy del Massachusetts Institute of Technology, y posteriormente de la Stanford University, goza de reconocimiento como la primera persona en darse cuenta de la importancia de la recursión en los lenguajes de programación. Él recomendó mucho su inclusión en el diseño de *Algol60* (un precursor de Pascal, PL/I y C) y desarrolló el lenguaje *Lisp*, que introdujo estructuras de datos recursivas junto con procedimientos y funciones recursivos. En este texto, las listas siguen el modelo de Lisp. El valor de la recursión se apreció durante el periodo de intenso desarrollo de algoritmos en los años setenta, y actualmente casi todos los lenguajes de programación populares apoyan la recursión.

La recursión y la inducción están íntimamente relacionadas. La presentación de la inducción en este capítulo procura dejar bien clara esa relación. En un sentido muy literal, una demostración por inducción puede considerarse como una demostración recursiva. La demostración de las propiedades de los procedimientos recursivos se simplifica mucho por su similitud estructural. (En este capítulo, al igual que en el anterior, incluiremos "función" en el significado general de "procedimiento"; la terminología de Java es "método".)

Presentaremos los árboles de recursión en la sección 3.7 para contar con un marco general en el cual analizar las necesidades de tiempo de los procedimientos recursivos. Se resolverán varios patrones de recursión que se encuentran con frecuencia, y los resultados se resumirán en teoremas.

3.2 Procedimientos recursivos

Entender con claridad cómo funciona realmente la recursión en la computadora ayuda mucho a pensar recursivamente, a ejecutar código recursivo a mano, y permite analizar el tiempo de ejecución de los procedimientos recursivos. Comenzaremos con un breve repaso de cómo se implementan las invocaciones de procedimientos con *marcos de activación*, y el apoyo que brindan las invocaciones a la recursión. Sin embargo, para la mayor parte de las actividades relacionadas con el diseño y análisis de procedimientos recursivos queremos pensar en un nivel más alto que el de los marcos de activación. A fin de ayudar a los lectores en este sentido, presentaremos el Método 99, que en realidad es un truco mental que permite diseñar soluciones recursivas.

3.2.1 Marcos de activación e invocaciones de procedimiento recursivas

En esta sección presentaremos una descripción breve y un tanto abstracta de cómo se implementan las invocaciones de procedimientos de modo que funcione la recursión. Si desea una descripción más exhaustiva, consulte las fuentes que se dan en las Notas y Referencias al final del capítulo.

La unidad básica de almacenamiento para una invocación de procedimiento individual durante la ejecución se denomina *marco de activación*. Este marco proporciona espacio para guardar las variables locales del procedimiento, los parámetros reales y las "variables temporales" del compilador, incluido el valor devuelto si es que el procedimiento devuelve un valor. También proporciona espacio de almacenamiento para otras necesidades contables, como la dirección de retorno, que indica cuál instrucción deberá ejecutar el programa una vez que salga de este procedimiento. Así, se crea un "marco de referencia" en el que el procedimiento se ejecuta *únicamente durante esta invocación*.

El compilador genera código para asignar espacio en una región de la memoria llamada *pila de marcos* (que a menudo se abrevia a sólo "pila"), como parte del código que implementa una

invocación de procedimiento. Se hace referencia a este espacio con un registro especial llamado *apuntador de marco*, de modo que, mientras se ejecuta esta invocación de procedimiento, se sepa dónde están almacenadas las variables locales, los parámetros de entrada y el valor devuelto. Cada invocación de procedimiento activa tiene un marco de activación único. Una investigación de procedimiento está activa desde el momento en que se entra en ella hasta que se sale de ella. Si hay recursión, todas las invocaciones del procedimiento recursivo que están activas simultáneamente tienen marcos distintos. Cuando se sale de una invocación de procedimiento (recursiva o no), su marco de activación se desocupa automáticamente para que alguna invocación de función futura pueda usar ese espacio. Una ejecución a mano de código que muestra los estados de los marcos de activación se denomina *rastreo de activación*.

Ejemplo 3.1 Marcos de activación para la función de Fibonacci

La figura 3.1 muestra varios puntos de un rastreo de activación para la función de Fibonacci, donde `main` ejecuta `x = fib(3)`. El pseudocódigo de `fib` es

```
int fib(int n)
    int f, f1, f2;
1.  if(n < 2)
2.    f = n;
3.  else
4.    f1 = fib(n − 1);
5.    f2 = fib(n − 2);
6.    f = f1 + f2;
7.  return f;
```

Este código declara algunas variables locales que normalmente serían variables temporales generadas por el compilador, con el fin de que veamos con mayor detalle el marco de activación. De hecho, la función `fib`, al igual que muchas funciones definidas recursivamente, se puede escribir en un solo enunciado "gigante", así:

```
return n < 2 ? n : fib(n−1) + fib(n−2);
```

pero esta forma no se presta al rastreo de activación.

La fila superior, columna izquierda, de la figura 3.1 muestra el marco de pilas justo antes de invocarse `fib(3)`, y la siguiente pila lo muestra inmediatamente después de que se entra en `fib`. La línea indicada bajo cada marco es la que está a punto de ejecutarse, o la que está a la mitad de su ejecución si ese marco no está en el tope de la pila de marcos. La ejecución del programa siempre está "en" el marco de activación de hasta arriba, de modo que las líneas que aparecen en otros marcos indican dónde estaba la ejecución cuando una investigación de procedimiento transfirió la ejecución a un nuevo marco de la pila. El valor de cada variable local aparece después del signo de dos puntos. Las variables que no tienen valores todavía no se han inicializado.

Las filas subsiguientes muestran el avance de la ejecución hasta la línea 4, donde hay otra invocación de función. (No importa que sea una invocación recursiva.) Para ahorrar espacio, en la siguiente fila se omite el avance de la línea 1 a la 4 y se limita a mostrar la línea 4 después de la siguiente invocación de función. Esta invocación avanza a la línea 2 y luego a la 7, porque se ha presentado un caso base; `f` ha recibido su valor y esta invocación está a punto de regresar. La última línea de la columna muestra la situación después de que la invocación anterior ha devuelto el valor 1; el valor devuelto se guardó como `f1` y está a punto de ejecutarse la línea 5.

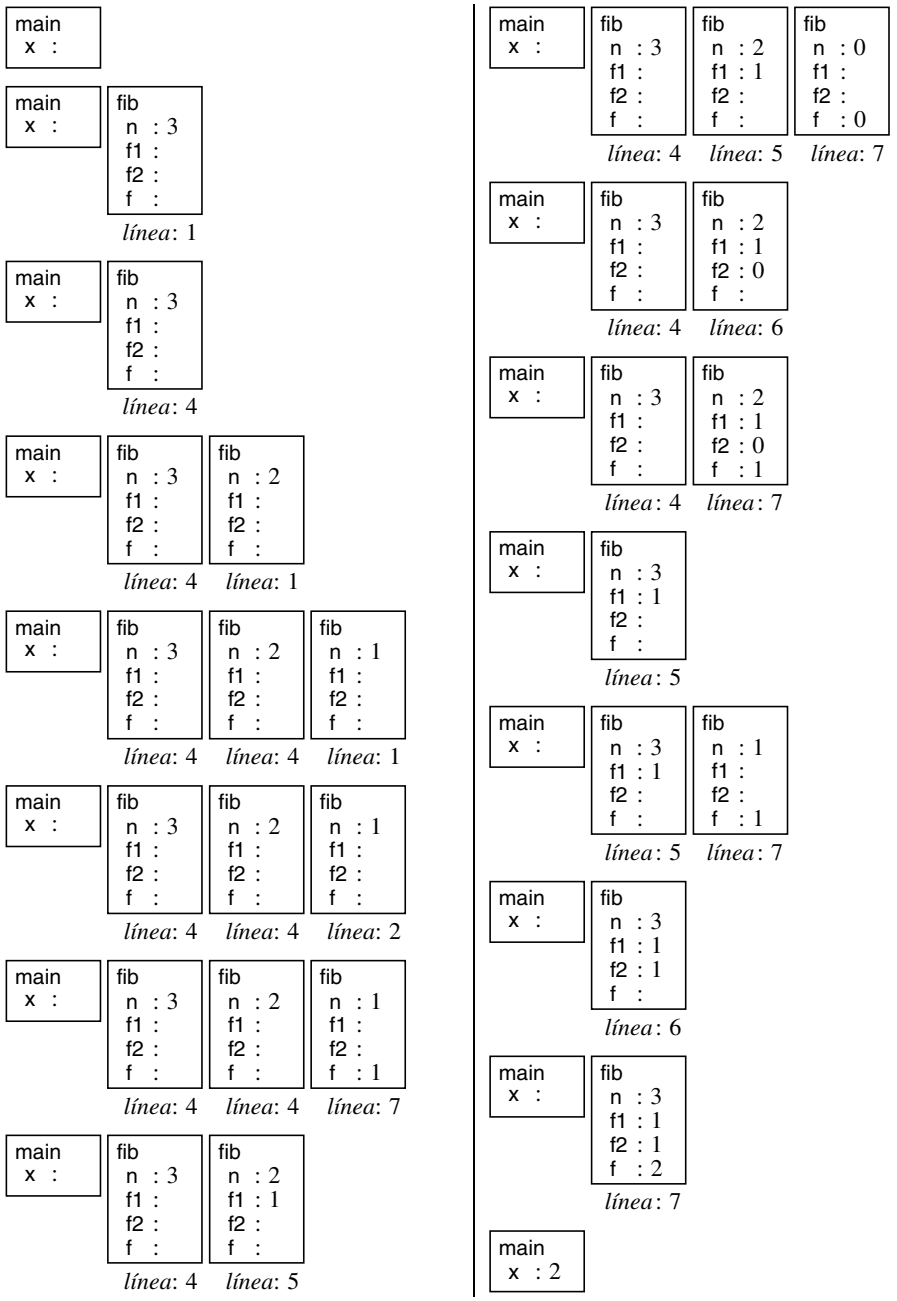


Figura 3.1 Rastreo de activación de la función `fib`: el tope de la pila está a la derecha. La sucesión de instantáneas baja por la columna izquierda y luego por la derecha.

La parte superior de la columna derecha muestra la situación después de que la invocación de `fib(0)` ha llegado a la línea 7; está a punto de regresar. Ahora se reutiliza el espacio del marco de activación que se liberó al terminar la invocación de `fib(1)`. Las tres filas que siguen muestran el final de la invocación `fib(2)`. El valor que devuelve se guarda en la copia de `f1` que está en el marco de activación para `fib(3)`. Este marco avanza hasta la línea 5. La siguiente invocación de función expande otra vez la pila. Luego la pila se contrae a medida que las invocaciones previas terminan su procesamiento y regresan. ■

Digamos que un *enunciado simple* de un procedimiento es cualquier enunciado que no invoca un procedimiento. Como ilustra el código anterior para `fib`, es posible escribir el procedimiento como una sucesión de líneas con un máximo de una invocación de procedimiento o enunciado simple por línea. Es razonable suponer que la ejecución de cada enunciado simple tarda un tiempo constante y que las actividades de contabilidad en torno a una invocación de procedimiento (preparar el siguiente marco de activación, etc.) también son constantes. Por tanto, se sigue que:

Lema 3.1 En un cálculo sin ciclos **`while`** o **`for`**, pero posiblemente con invocaciones de procedimiento recursivas, el tiempo durante el que cualquier marco de activación dado está en el tope de la pila de marcos es $O(L)$, donde L es el número de líneas del procedimiento que contienen un enunciado simple o bien una invocación de procedimiento. □

Sin embargo, el tamaño L de cualquier procedimiento también es constante; es decir, no cambia las entradas. En cualquier algoritmo fijo existe una L máxima para todos los procedimientos de ese algoritmo. El tiempo total que tarda cualquier ejecución dada del algoritmo es por fuerza la suma de los tiempos que los diversos marcos de activación pasan en el tope de la pila de marcos. También es razonable suponer que cualquier marco de activación que se coloca en la pila pasa en ella cierto tiempo mínimo, debido a las actividades de contabilidad, aunque regrese "instantáneamente". Esto nos proporciona una herramienta potente para analizar el tiempo de ejecución de un cálculo recursivo.

Teorema 3.2 En un cálculo sin ciclos **`while`** o **`for`**, pero posiblemente con invocaciones de procedimiento recursivas, el tiempo de cálculo total es $\Theta(C)$, donde C es el número total de invocaciones de procedimientos (incluyendo las invocaciones de funciones como invocaciones de procedimientos) que se efectúan durante el cálculo. □

Para llevar esta idea un paso más lejos, podemos definir un *árbol de activación* para crear un registro permanente de todas las invocaciones de procedimientos que se efectuaron durante una ejecución de un algoritmo. Cada nodo corresponde a una invocación de procedimiento distinta, justo en el punto en que está a punto de regresar. La raíz es la invocación de nivel más alto en ese algoritmo. El padre de cada tercer nodo es simplemente el nodo cuyo marco de activación estaba en el tope de la pila de marcos en el momento en que se creó ese nodo. Los hijos de cada nodo aparecen de izquierda a derecha en el orden en que se crearon sus marcos de activación. En la figura 3.2 se da un ejemplo.

Un recorrido en orden previo del árbol de activación visita cada marco de activación en orden cronológico de creación, y el número de nodos del árbol es proporcional al tiempo de ejecución total. Cualquier instantánea del marco de pilas durante la ejecución corresponderá a algún camino de este árbol que parta desde la raíz. (Volveremos a esta correspondencia cuando hable-

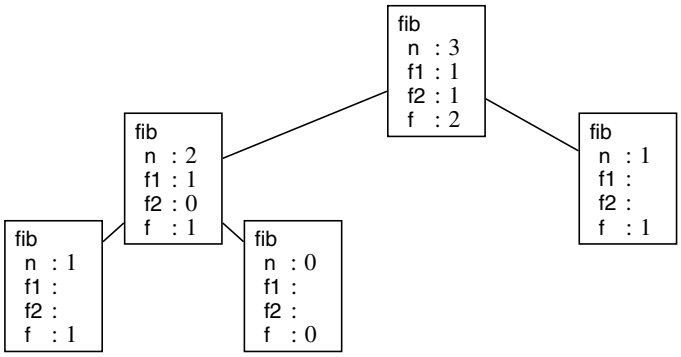


Figura 3.2 Árbol de activación para `fib(3)`

mos de la búsqueda de primero en profundidad en la sección 7.4.1.) En la sección 3.7.3 examinaremos una relación entre los árboles de activación y el análisis de ecuaciones de recurrencia, la cual es muy útil para analizar algoritmos recursivos.

3.2.2 Sugerencias para recursión: Método 99

En el desarrollo avanzado de algoritmos, la recursión es una técnica de diseño indispensable. Un tratamiento a fondo del diseño recursivo rebasa el alcance de este libro, pero sí presentaremos unas cuantas sugerencias. En las Notas y referencias al final del capítulo se sugieren lecturas adicionales.

Identificamos alguna "unidad de medida" para el tamaño del problema que nuestra función o procedimiento tratará de resolver. Luego imaginamos que nuestra tarea consiste en escribir un procedimiento, digamos `p`, que resolverá problemas de *todos los tamaños entre 0 y 100*. Esto implica que, al diseñar la solución, podremos *suponer* que el tamaño del problema es cuando más 100: ésta es nuestra "condición previa fantasiosa".

Además, imaginamos que se nos permite invocar una *subrutina dada*, de nombre `p99`, que hace exactamente lo que se supone que hace nuestro procedimiento, y tiene la misma rúbrica de tipo, con la excepción de que *su* "condición previa fantasiosa" es que el tamaño de su problema es de 0 a 99. Podemos usar esta subrutina (a condición de que se le invoque con parámetros que satisfagan sus condiciones previas) sin tener que escribir el código correspondiente.

Una segunda sugerencia es identificar claramente el caso no recursivo del problema. Conviene hacerlo lo más pequeño posible. Nuestro procedimiento casi siempre iniciará con una prueba de este caso no recursivo, también llamado *caso base*.

Una última estipulación es que resulta "demasiado costoso" determinar si el problema alimentado a `p` tiene un tamaño de exactamente 100. (Podríamos haber usado como límite fantasioso un tamaño de 1,000,000,000, pero habría sido muy latoso decir "método 999,999,999".) En cambio, es factible determinar si su tamaño es 0, o cualquier constante pequeña.

Ahora bien, el *Método 99* consiste en encontrar una forma de escribir `p` invocando a `p99` siempre que se necesite. (No es necesario escribir `p99`, así que nos olvidamos de él.) Desde luego, si `p` detecta un caso fácil, no *necesitará* invocar a `p99`. La idea clave es que, cuando `p` detecta un caso que no se puede resolver de inmediato, tiene que crear un subproblema para resolverlo con `p99`, el cual satisface tres condiciones:

1. El tamaño del subproblema es menor que el del problema de `p`.
2. El tamaño del subproblema no es menor que el mínimo (0, en esta explicación).
3. El subproblema satisface todas las demás condiciones previas de `p99` (que son iguales a las condiciones previas de `p`).

Se garantiza (en nuestra fantasía) que el subproblema satisface las restricciones de tamaño de `p99` (¿por qué?).

Si podemos descomponer la solución de esta manera, ya casi terminamos. Bastará con escribir el código de `p`, invocando a `p99` siempre que sea necesario.

Practiquemos con la tarea de escribir `borrar(L, x)`, que supuestamente borra el elemento `x` de una `ListaInt` `L`, devolviendo una nueva `ListaInt` que contiene todos los elementos de `L` con excepción de la primera ocurrencia de `x`. Es posible que `L` no contenga `x`. El tamaño del problema es el número de elementos de la lista `L`. (El TDA se describe en la sección 2.3.2; el constructor es `cons`, las funciones de acceso son `primero` y `resto`, y la constante `nil` denota la lista vacía.)

Para aplicar el Método 99, imaginamos que sólo tenemos que preocuparnos por listas que contienen 100 elementos o menos, y que se nos da `borrar99`. Es evidente que, si podemos eliminar un elemento (digamos, el primero) de `L`, podremos dejar que `borrar99` se encargue de `resto(L)`. No sabemos cuántos elementos hay en `resto(L)`, pero adoptamos una actitud intransigente: si hay cuando más 100 elementos en `L`, podemos invocar `borrar99`, y si hay más, no nos importa qué suceda porque (en nuestra fantasía) sólo se nos pidió hacer que `borrar` funcionara con una lista de 100 elementos o menos.

Siguiendo la segunda sugerencia, necesitamos probar el caso base. ¿Cuál es el caso base?, puesto que se permite que `x` no esté en la lista, podemos tener una lista vacía. Además de la lista vacía, hay otro caso que podemos reconocer y resolver instantáneamente, sin tener que usar `borrar99`: si `x` es el primer elemento de `L`. En este caso, cumplimos con el cometido de `borrar` con sólo devolver `resto(L)`.

Así pues, hemos llegado al siguiente procedimiento de Método 99 para implementar `borrar`.

```
ListaInt borrar(ListaInt L, int x)
    ListaInt nuevaLista, listaFija;
    if (L == nil)
        nuevaLista = L;
    else if (x == primero(L))
        nuevaLista = resto(L);
    else
        listaFija = borrar99(resto(L), x);
        nuevaLista = cons(primero(L), listaFija);
    return nuevaLista;
```

Ah, claro. Para terminar, basta con quitar el "99" del nombre de la subrutina invocada, convirtiéndola en una invocación recursiva del mismo procedimiento.

El procedimiento `borrar` es otro de los que encajan en el patrón de las *rutinas de búsqueda generalizada* (véase la definición 1.12): si no hay más datos, fracasar; si este dato es el que estamos buscando, tener éxito (borrándolo, en este caso); de lo contrario, continuar la búsqueda en los datos restantes.

3.2.3 Envolturas para procedimientos recursivos

Es común que una tarea tenga partes que sólo deban ejecutarse una vez al principio o al final. En tales casos, se necesita un procedimiento no recursivo que prepare las cosas y luego invoque al procedimiento recursivo, y tal vez efectúe tareas finales una vez que ese procedimiento regrese. ("Procedimiento" incluye las funciones.) Llamamos a un procedimiento no recursivo de este tipo *envoltura* del procedimiento recursivo. A veces la envoltura se limita a inicializar un argumento adicional para el procedimiento recursivo. Por ejemplo, Búsqueda Binaria (algoritmo 1.4) necesita una envoltura para efectuar la primera invocación con todo el arreglo como intervalo. La envoltura puede ser simplemente

```
int busquedaOrdenada(int[] E, int n, int K)
     return busquedaBinaria(E, 0, n−1, K);
```

3.3 ¿Qué es una demostración?

Antes de pasar a las pruebas por inducción, dediquemos un momento a repasar la naturaleza de las demostraciones. Como mencionamos en la sección 1.3.3, la lógica es un sistema para formalizar afirmaciones hechas en el lenguaje natural para poder razonar con mayor exactitud. Las demostraciones son el *resultado* de razonar con enunciados lógicos. En esta sección describiremos demostraciones detalladas. En la práctica es común omitir muchos detalles y dejar que el lector llene los huecos; tales explicaciones son más bien *bosquejos de demostración*.

Los teoremas, lemas y corolarios son enunciados susceptibles de demostrarse, y las diferencias no están definidas con precisión. En general, un lema es un enunciado que no resulta muy interesante por sí solo, pero que es importante porque ayuda a demostrar algo que *sí es* interesante y que normalmente se clasifica como teorema. Un corolario por lo regular es una consecuencia directa de un teorema, pero no necesariamente es menos importante. No importa si el enunciado a demostrar se denomina "propuesta", "teorema", "lema", "corolario" u otra cosa, el proceso de demostración es el mismo. Utilizaremos "propuesta" como término genérico.

Una demostración es una sucesión de *enunciados* que forman un argumento lógico. Cada enunciado es una *oración completa* en el sentido gramatical usual: tiene sujeto, verbo y complementos. Aunque la notación matemática permite abreviarlos, los enunciados deben corresponder a una oración completa. Por ejemplo, "$x = y + 1$" corresponde a "x es igual a $y + 1$", que es una oración completa, mientras que "$y + 1$" por sí sola no es una oración.

Aunque es posible presentar una lista exhaustiva de las *reglas de inferencia* precisas para combinar enunciados lógicos para producir una demostración, adoptaremos un enfoque más informal. Las reglas más importantes se dan en la sección 1.3.3, ecuaciones 1.29 a 1.31. Cada enunciado deberá sacar una conclusión nueva de hechos que son

- bien conocidos, y no lo que estamos tratando de demostrar (por ejemplo, identidades matemáticas), o
- supuestos (premisas) del teorema que estamos demostrando, o
- enunciados establecidos previamente en la demostración (conclusiones intermedias), o
- ejemplares de la *hipótesis inductiva*, que veremos en la sección 3.4.1.

El último enunciado de una demostración debe ser la conclusión de la propuesta que se está demostrando. Cuando una demostración se ramifica en varios casos, cada caso deberá tener la estructura anterior.

Cada enunciado deberá plantear no sólo la nueva conclusión, sino en qué se apoya: los hechos de los que depende. Los enunciados que apoyan de forma inmediata la nueva conclusión son sus *justificaciones*. Las justificaciones vagas son la causa de la mayor parte de los errores de lógica.

Formato del teorema o propuesta

La propuesta que necesitamos demostrar tiene dos partes, los *supuestos* (también llamados *premisas* o *hipótesis*) y la *conclusión*. Llamemos a la conclusión *enunciado meta*. Por lo regular, la propuesta tiene una frase de la forma "para toda x en el conjunto W", y el enunciado meta dice algo acerca de x. (Podría haber más de una variable como x en los enunciados.) En la práctica, el conjunto W (el "mundo") es algún conjunto conocido, como los números naturales, los reales o una familia de estructuras de datos, como listas, árboles o grafos. Representemos de forma abstracta la propuesta a demostrar como

$$\forall x \in W \big[A(x) \Rightarrow C(x)\big]. \tag{3.1}$$

Aquí, $A(x)$ representa los supuestos y $C(x)$ representa la conclusión, o enunciado meta. El símbolo "$\Rightarrow$" se lee "implica". Los corchetes sólo sirven para hacer más comprensible la notación; agrupan igual que los paréntesis. En lenguaje natural, el enunciado propuesta suele adoptar la forma "para toda x en W, si $A(x)$, entonces $C(x)$". Puede haber muchas variaciones en las palabras. En muchos casos es necesario modificar el enunciado más natural de modo que se ajuste más o menos a una forma estándar como en los ejemplos anteriores y así tener la certeza de qué es lo que queremos demostrar, y qué partes corresponden a x, W, $A(x)$ y $C(x)$.

Ejemplo 3.2

Una propuesta podría expresarse así:

Propuesta 3.3 Para las constantes $\alpha < \beta$, $2^{\alpha n} \in o(2^{\beta n})$. □

Si lo replanteamos siguiendo el formato general de la ecuación (3.1), se convierte en

Propuesta 3.4 Para toda $\alpha \in \mathbf{R}$, para toda $\beta \in \mathbf{R}$, si α y β son constantes y $\alpha < \beta$, entonces $2^{\alpha n} \in o(2^{\beta n})$. □

Verifiquemos las correspondencias. Vemos que el par (α, β) hace las veces de x y que $\mathbf{R} \times \mathbf{R}$ hace las veces de W. La hipótesis del teorema, $A(\alpha, \beta)$, son los tres enunciados "α es constante", "β es constante" y "$\alpha < \beta$". La conclusión, $C(\alpha, \beta)$, es "$2^{\alpha n} \in o(2^{\beta n})$".

Formato de demostración de dos columnas

A continuación describiremos un formato de dos columnas para presentar las demostraciones. El objetivo de este formato es aclarar el papel de las justificaciones en la demostración; la columna de la derecha contiene todas las justificaciones. Cada enunciado de la demostración ocupa una línea numerada. Cada *conclusión nueva* de la columna izquierda se aparea con sus justificaciones de la columna derecha. Se hace referencia a enunciados anteriores de la demostración dando sus números de línea.

Ejemplo 3.3

El enunciado del ejemplo 3.2 se demuestra en el formato de dos columnas. Tanto el teorema como la demostración se han redactado de forma poco concisa, como ilustración de la forma en que las justificaciones encajan en la demostración. Se han incluido todas las partes que los autores normalmente suponen serán aportadas por los lectores. Haremos algunos comentarios más después de la demostración.

Teorema 3.5 Para toda $\alpha \in \mathbf{R}$, para toda $\beta \in \mathbf{R}$, si α y β son constantes, con $\alpha < \beta$, entonces $2^{\alpha n} \in o(2^{\beta n})$.

Demostración

	Enunciado	Justificación
1.	Primero queremos demostrar que $\lim_{n\to\infty} \frac{2^{\beta n}}{2^{\alpha n}} = \infty$.	
2.	$\frac{2^{\beta n}}{2^{\alpha n}} = 2^{(\beta-\alpha)n}$.	Identidad matemática.
3.	$\beta - \alpha > 0$ y es constante.	Hipótesis del teorema + identidad matemática.
4.	$\lim_{n\to\infty} 2^{(\beta-\alpha)n} = \infty$.	(3) + propiedad matemática conocida.
5.	$\lim_{n\to\infty} \frac{2^{\beta n}}{2^{\alpha n}} = \infty$.	(2) + (4) y sustitución.
6.	$2^{\alpha n} \in o(2^{\beta n})$.	(5) + definición de conjuntos o (definición 1.17). □

Algunos comentarios adicionales:

1. Además de los enunciados que constituyen el meollo de la demostración, es común incluir algunos enunciados de "croquis" o "plan" que indiquen lo que se supone que una sección de la demostración pretende mostrar, o qué metodología general se usará, o qué falta demostrar para terminar una sección de la demostración, etcétera.

   Es evidente que la línea 1 se planteó como enunciado de *plan*, no como conclusión. Por tanto, no podemos referirnos a ella posteriormente y no requiere justificación; le dice al lector cuál es la meta intermedia de las líneas subsiguientes de la demostración. Esa meta se alcanza en la línea 5.
2. La conclusión nueva de la última línea es exactamente el enunciado meta del teorema.
3. Se hace referencia a *todas* las demás líneas como justificación; nada se desperdicia. ■

Para familiarizarse con las demostraciones y adquirir fluidez, es recomendable escribir algunas con todos sus detalles, siguiendo el formato de dos columnas. Casi siempre se aprende algo al detallar los bosquejos de demostración, asegurándose de entender cómo se concluye cada enunciado nuevo.

3.4 Demostraciones por inducción

Las demostraciones por inducción son un mecanismo, a menudo el *único*, para demostrar una afirmación acerca de un conjunto infinito de objetos. El método de inducción que describimos aquí suele clasificarse como inducción *fuerte*. La inducción fuerte es la forma más fácil de usar en la mayor parte de las demostraciones relacionadas con algoritmos y estructuras de datos. Aunque no se necesite toda su potencia, no es más difícil de usar que sus variantes más débiles. Por ello, adoptaremos un enfoque "unitalla" y sólo utilizaremos este método.

Más adelante veremos que la recursión y la inducción (fuerte) se complementan. En general, las demostraciones son difíciles, y todo lo que pueda ayudarnos a hacerlas comprensibles y siempre exactas es bienvenido. La similitud de estructura entre las demostraciones por inducción y los procedimientos recursivos es un apoyo importante para analizar algoritmos avanzados.

En muchos casos, la inducción se aplica al conjunto de los números naturales (enteros no negativos, sección 1.5.1) o el conjunto de los enteros positivos. Sin embargo, el método de inducción es válido para conjuntos más generales, siempre que posean dos propiedades:

1. El conjunto está parcialmente ordenado; es decir, se define una relación de orden entre algunos pares de elementos, pero quizá no entre todos los pares.
2. No existe alguna cadena infinita de elementos decrecientes en el conjunto.

Por ejemplo, no podemos usar inducción con el conjunto de *todos* los enteros (con el orden acostumbrado).

Los árboles son un ejemplo de conjunto parcialmente ordenado que se utiliza a menudo para la inducción. El orden parcial acostumbrado se define como $t_1 < t_2$ si t_1 es un *subárbol propio* de t_2 (véase la figura 3.3). Más adelante veremos que los grafos pueden tener también un orden parcial similar. La inducción con este tipo de conjuntos se conoce como *inducción estructural*.

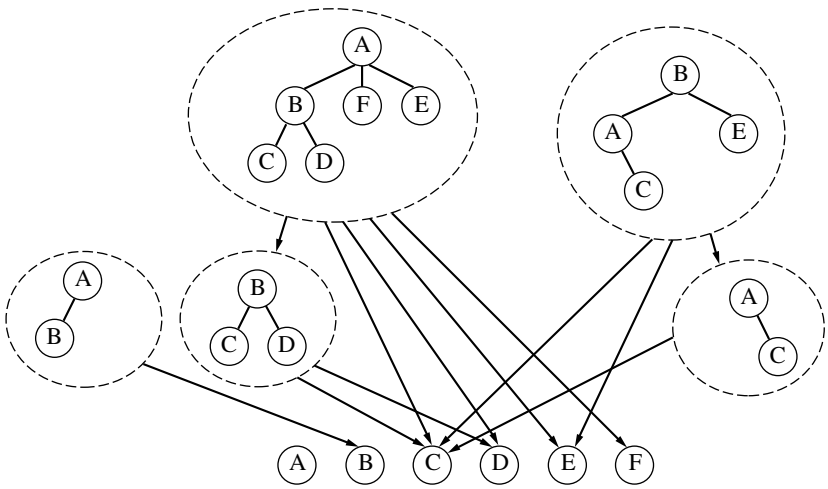


Figura 3.3 El orden parcial *subárbol* entre el conjunto de árboles que se muestra

Los teoremas que suelen requerir una demostración por inducción son los que se refieren a una fórmula matemática, a una propiedad de un procedimiento, a una propiedad de las estructuras de datos y a ecuaciones de recurrencia, que surgen con frecuencia durante el análisis del tiempo de ejecución de un procedimiento recursivo. En la sección 3.5 se tratarán los tipos de lemas que se requieren para demostrar que un procedimiento logra sus objetivos y termina. La sección 3.6 trata las ecuaciones de recurrencia típicas.

3.4.1 Esquema de una demostración por inducción

Lo primero que debemos entender acerca de una demostración por inducción es

No existe "$n + 1$" en una demostración por inducción.

Lamentablemente, a muchos lectores se les enseñó lo contrario. ¿Por qué estamos adoptando esta postura tan dogmática?

La respuesta radica en el motivo que planteamos antes: relacionar las demostraciones por inducción con procedimientos recursivos. Sabemos que un procedimiento recursivo funciona *creando y resolviendo subproblemas más pequeños*, y combinando luego las soluciones más pequeñas para resolver el problema principal. Queremos que nuestra demostración por inducción siga este plan. Para la demostración, el "problema principal" es el teorema planteado, y los "subproblemas" son casos más limitados del teorema planteado que se pueden combinar para demostrar el caso principal. En la práctica, es muy probable que exista una correspondencia directa entre esos casos y los subproblemas exactos creados por el procedimiento recursivo.

Todas las demostraciones por inducción siguen un patrón común, que llamaremos *esquema de inducción*. La parte más importante del esquema es la introducción correcta de la *hipótesis inductiva*. Primero presentaremos un ejemplo de demostración, luego describiremos el esquema general y terminaremos con más ejemplos.

Ejemplo 3.4

La demostración de la propuesta siguiente ilustra el esquema de inducción que describiremos en forma general después de este ejemplo. Las frases en **negrita** son elementos que aparecen prácticamente al pie de la letra en cualquier demostración por inducción detallada. Después de la demostración, que por claridad se ajusta al formato de dos columnas, vienen comentarios pormenorizados.

Propuesta 3.6 Para toda $n \geq 0$, $\sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{n(n+1)(n+2)}{6}$.

Demostración

	Enunciado	Justificación
1.	**La demostración es por inducción con** n, el límite superior de la sumatoria.	
2.	**El caso base es** $n = 0$.	
3.	En este caso ambos miembros de la ecuación son 0.	Matemáticas.

4. **Para n mayor que 0, suponemos que**

$$\sum_{i=0}^{k} \frac{i(i+1)}{2} = \frac{k(k+1)(k+2)}{6}$$

se cumple para toda $k \geq 0$ tal que $k < n$.

5. $\displaystyle\sum_{i=0}^{n-1} \frac{i(i+1)}{2} = \frac{(n-1)(n)(n+1)}{6}$. Hip. Ind. con $k = n - 1$.

6. $\displaystyle\sum_{i=1}^{n} \frac{i(i+1)}{2} = \sum_{i=1}^{n-1} \frac{i(i+1)}{2} + \frac{n(n+1)}{2}$. Matemáticas.

7. $\displaystyle\sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{(n-1)(n)(n+1)}{6} + \frac{n(n+1)}{2}$. (5) + (6).

8. $\displaystyle\frac{(n-1)(n)(n+1)}{6} + \frac{n(n+1)}{2} = \frac{n(n+1)(n+2)}{6}$. Matemáticas.

9. $\displaystyle\sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{(n+1)(n)(n+2)}{6}$. (7) + (8). □

He aquí comentarios línea por línea.

1. Anunciamos que n es la principal variable de inducción. Observe que la propuesta tiene la forma $\forall n \in \mathbf{N}\ [A(n) \Rightarrow C(n)]$. Aquí, $A(n)$ es simplemente el valor booleano *verdadero*, y $C(n)$ es la ecuación.
2. Una demostración por inducción siempre tiene dos casos principales, llamados *caso base* y *caso inductivo*. Identificamos el o los casos base.
3. *Demostramos* el o los casos base.
4. Introducimos la variable auxiliar k y planteamos la hipótesis inductiva. Observe que la hipótesis inductiva adopta la forma $A(k) \Rightarrow C(k)$, recordando que en el caso de esta propuesta $A(k)$ es simplemente el valor booleano *verdadero*.

   Observe que el intervalo de k incluye los casos base.

   Observe que k sólo puede ser estrictamente menor que n; de lo contrario, estaríamos suponiendo lo que estamos tratando de demostrar.

   El planteamiento de la hipótesis inductiva indica que aquí comienza la demostración del caso inductivo.
5. *Usamos* la hipótesis inductiva. (Observe que estamos "acercándonos a" n.) La variable auxiliar k se iguala *localmente* a $n - 1$. Puesto que estamos demostrando el caso en que $n > 0$, este valor de k satisface $0 \leq k < n$, como exige la línea 4.

   La variable auxiliar k podría igualarse a otros valores dentro de su intervalo en otras líneas, si la demostración lo requiere. Ésta es una ventaja de la inducción "fuerte". En esta demostración sencilla da la casualidad que no se requieren otras igualaciones de k.

6. La justificación es una identidad matemática estándar que seguramente el lector conoce.
7. La justificación indica las dos líneas anteriores que apoyan la nueva conclusión, pero no dice cuál *regla de inferencia* se usó, pues se supone que el lector puede deducirla. En este caso la regla de inferencia es la conocida como "sustitución de iguales por iguales", o simplemente "sustitución". La línea 9 es similar.
8. Se aplica otra identidad matemática. En la práctica, las líneas 6 a 9 se condensarían en una sola, suponiéndose que el lector puede deducir los pasos. Sin embargo, tales condensaciones dan pie o contribuyen a muchas "demostraciones" erróneas. Quien escribe la demostración debe cuidar que *pueda* escribirse una serie precisa de pasos.
9. Esta conclusión es exactamente el enunciado meta, $C(n)$. ■

La demostración anterior sigue un patrón que se puede generalizar al siguiente esquema. Cabe señalar que el término genérico *propuesta* puede ser *teorema*, *lema*, *corolario* o algún otro término, sin que tenga que alterarse el proceso de demostración.

Definición 3.1 Esquema de demostración por inducción

Primero explicaremos la notación empleada en el esquema. El texto en **negritas** aparece prácticamente letra por letra. Los términos encerrados en paréntesis angulares, "⟨, ⟩", se reemplazan por sustitución, según la propuesta a demostrar. Así mismo, las variables x y y adoptan nombres acordes con la propuesta, y su intervalo es el conjunto W (el "mundo"). El enunciado lógico $C(x)$ es el *enunciado meta*. El enunciado lógico $A(x)$ es la *hipótesis de la propuesta* (o las hipótesis, si es una conjunción). La variable x es la *variable de inducción principal* (o simplemente *variable de inducción*). La variable y es la *variable auxiliar*.

Una demostración por inducción de una propuesta con la forma

$$\forall x \in W\, [A(x) \Rightarrow C(x)]$$

consta de las partes siguientes, en ese orden.

1. **Las demostración es por inducción con** x, ⟨descripción de x⟩.
2. **El caso base es (los casos base son)** ⟨caso base⟩.
3. ⟨*Demostración* del enunciado meta en el que se ha sustituido el caso base, es decir, C(caso base).⟩
4. **Para** ⟨x⟩ **mayor que** ⟨caso base⟩, **suponemos que se cumple** $[A(y) \Rightarrow C(y)]$ **para toda** $y \in W$ **tal que** $y < x$.
5. ⟨*Demostración* del enunciado meta, $C(x)$, exactamente como aparece en la propuesta.⟩ ■

Una prueba por inducción tiene dos casos principales: el caso base y el caso inductivo. La parte (2) del esquema define el caso base; la parte (3) demuestra el teorema para el caso base, que podría estar dividida en varios casos. La parte (4) define el caso inductivo, y plantea la hipótesis inductiva. La parte (5) demuestra el teorema para el caso inductivo, y por lo regular es el meollo de la demostración. Esta demostración de $C(x)$ se puede apoyar con:

1. el hecho de que x es mayor que ⟨caso base⟩ en este caso de la demostración;
2. la hipótesis de la propuesta, $A(x)$ (pero *no* $A(y)$);

3. cualquier cantidad de ejemplos de la hipótesis inductiva, que es $[A(y) \Rightarrow C(y)]$, sustituyendo por la variable auxiliar y elementos de W que son *estrictamente menores que* x.

Como siempre, se pueden usar conclusiones anteriores de la demostración, identidades externas, teoremas y demás.

Se permiten tres enunciados "de machote" sin tener que justificarlos, pues no sacan conclusiones; simplemente explican el esquema de la demostración y definen cierta notación. Ellos son

- "La demostración es por inducción con x, ..."
- "El caso base es ..."
- "Para $x >$ ⟨caso base⟩, suponemos que se cumple $[A(y) \Rightarrow C(y)]$ para toda $y < x$."

Los últimos dos enunciados dividen la demostración en dos casos: x es un caso base, y x es mayor que cualquier caso base. Estos dos casos deberán cubrir todo el conjunto W del cual x toma valores.

Variaciones del esquema de inducción

1. Si los supuestos $A(x)$ no dependen realmente de x, la hipótesis inductiva se simplifica a: **Suponemos que se cumple** $C(y)$ **para toda** $y \in W$ **menor que** x. El lector deberá poder explicar por qué se justifica esta simplificación remitiéndose a las justificaciones de los enunciados de una demostración.
2. Podría haber dos o más casos base si el caso inductivo requiere más de un caso menor, como con los números de Fibonacci, ecuación (1.13). Sin embargo, lo mejor es colocar el mayor número posible de elementos en el caso inductivo, porque cada caso base requiere su propia demostración.
3. Podría haber muchos elementos de caso base si la inducción es con estructuras de datos, como listas, árboles o grafos, u otros conjuntos W que sólo tienen un orden parcial. En la figura 3.3 los seis árboles de un solo elemento son casos base.

3.4.2 Demostración por inducción de un procedimiento recursivo

El ejemplo que sigue muestra cómo operan juntas la inducción y la recursión. El lema que demostraremos, relativo a un procedimiento para calcular la longitud de un camino externo en árboles-2, es útil en el análisis de cotas inferiores (véase la sección 4.7.3). Las longitudes de caminos externos surgen naturalmente en varios otros problemas. Primero necesitamos varias definiciones.

Definición 3.2 Nodos externos y árboles-2

En ciertos tipos de árboles binarios, el caso base, en lugar de ser un árbol vacío, es un árbol con un solo nodo de un tipo distinto del resto del árbol. Este tipo de nodo se llama *nodo externo*. Un árbol que consiste en un nodo externo se llama *hoja*, y no tiene subárboles. El otro tipo de nodo se denomina *nodo interno*, y debe tener dos hijos. Tales árboles binarios se llaman *árboles-2* porque cada nodo tiene dos hijos o ninguno. ■

Cabe señalar que, si sustituimos todos los nodos externos de un árbol-2 por árboles vacíos, nos queda un árbol binario normal, sin restricciones. En un árbol-2 casi siempre es posible reconocer una hoja sin verificar si tiene hijos o no porque su nodo es de un tipo distinto del de los nodos internos. El Ejercicio 3.1 muestra que un árbol-2 debe tener un nodo externo más que nodos internos.

Definición 3.3 Longitud de camino externo

En un árbol-2 *t*, la *longitud de camino externo* de *t* es la suma de las longitudes de todos los caminos desde la raíz de *t* hasta cualquier nodo externo de *t*. La longitud de un camino es el número de aristas que incluye.

La *longitud de camino externo* de un árbol-2 también puede definirse inductivamente como sigue:

1. La longitud de camino externo de una hoja es 0.
2. Sea *t* un árbol que no es hoja, con subárbol izquierdo *L* y subárbol derecho *R* (cualquiera de los cuales puede ser una hoja). La longitud de camino externo de *t* es la longitud de camino externo de *L* más el número de nodos externos de *L* más la longitud de camino externo de *R* más el número de nodos externos de *R*. (El número de nodos externos de *t* es la suma de los números de nodos externos de *L* y de *R*.)

La equivalencia de las dos definiciones es obvia porque todos los caminos desde la raíz de *t* hasta un nodo externo de *L* tienen una arista más que el camino correspondiente desde la raíz de *L* hasta el mismo nodo externo, y lo mismo para *R*. ■

El esqueleto para recorrer un árbol binario de la sección 2.3.3 se puede adaptar fácilmente para calcular las longitudes del camino externo. La clase del parámetro es `ArbolDos`, que se define de forma análoga a `ArbolBin`, excepto que el árbol más pequeño es una hoja, no un árbol vacío. El caso base se modifica de manera acorde. La función debe devolver dos valores, por lo que suponemos que se ha definido una clase organizadora (véase la sección 1.2.2), `LceDevuelta`, que tiene dos campos enteros, `lce` y `numExt`, los cuales representan la longitud de camino externo y el número de nodos externos, respectivamente. El resultado se muestra en la figura 3.4. Vemos que la función se limita a implementar la versión inductiva de la definición. Ahora podemos demostrar el lema siguiente acerca de `calcLce`.

Lema 3.7 Sea *t* cualquier árbol-2. Sean *lce* y *m* los valores de los campos `lce` y `numExt`, respectivamente, devueltos por `calcLce`(*t*). Entonces:

1. *lce* es la longitud de camino externo de *t*.
2. *m* es el número de nodos externos de *t*.
3. $lce \geq m \lg(m)$.

Demostración Antes de demostrar el lema, correlacionemos el planteamiento del lema con nuestro patrón para las propuestas a demostrar, ecuación (3.1). Cabe señalar que lo hemos dividido en varias oraciones, para hacerlo más comprensible, pero sin omitir parte alguna. Así pues, *t* es la variable de inducción principal y *W* es el conjunto de todos los árboles-2. La segunda oración plantea las hipótesis, así que corresponde a $A(t)$. Por último, las tres conclusiones constituyen $C(t)$. Al igual que en ejemplos anteriores, el texto en **negritas** aparece prácticamente letra por letra en cualquier demostración por inducción.

```
LceDevuelta calcLce(ArbolDos t)
    LceDevuelta respL, respR;  // devueltas de los subárboles
    LceDevuelta resp = new LceDevuelta(); // para devolver

1.  if (t es una hoja)
2.      resp.lce = 0; resp.numExt = 1;
3.  else
4.      respL = calcLce(subarbolIzq(t));
5.      respR = calcLce(subarbolDer(t));
6.      resp.lce = respL.lce + respR.lce + respL.numExt + respR.numExt;
7.      resp.numExt = respL.numExt + respR.numExt;
8.  return resp;
```

Figura 3.4 Función para calcular la longitud de camino externo de un árbol-2. Se usa el tipo devuelto `LceDevuelta` para que la función pueda devolver dos cantidades, `lce` y `numExt`.

La demostración es por inducción con t, el parámetro de `calcLce`, con el orden parcial de "subárbol". **El caso base es** que t es una hoja. Se llega a la línea 2 de `calcLce`, así que $lce = 0$ y $m = 1$, lo cual es correcto para las partes (1) y (2), y se cumple $0 \geq 0$ para la parte (3).

Para t no hoja, **suponemos que se cumple el lema para toda** s, **donde** s es un subárbol propio de t. Es decir, si `calcLce`(s) devuelve lce_s y m_s, entonces m_s es el número de nodos externos de s, lce_s es la longitud de camino externo de s, y $lce_s \geq m_s \lg(m_s)$. Denotemos con L y R los subárboles izquierdo y derecho de t, respectivamente. Éstos son subárboles propios de t, así que es válida la hipótesis inductiva. Puesto que t no es una hoja, se ejecutan las líneas 4 a 7, de lo que se sigue que

$$\begin{aligned} lce &= lce_L + lce_R + m_L + m_R, \\ m &= m_L + m_R. \end{aligned}$$

Por la hipótesis inductiva y la definición inductiva de la longitud de camino externo, lce es la longitud de camino externo de t. Todo nodo externo de t está en L o bien en R, de modo que m es el número de nodos externos de t.

Falta demostrar que $lce \geq m \lg(m)$. Observamos (véase el ejercicio 3.2) que la función $x \lg(x)$ es convexa para $x > 0$, así que podemos usar el lema 1.3. Por la hipótesis inductiva tenemos

$$lce \geq m_L \lg(m_L) + m_R \lg(m_R) + m$$

$$m_L \lg(m_L) + m_R \lg(m_R) \geq 2\left(\frac{m_L + m_R}{2}\right) \lg\left(\frac{m_L + m_R}{2}\right).$$

Entonces, por la transitividad de "≥",

$$lce \geq m\,(\lg(m) - 1) + m = m \lg(m). \qquad \square$$

Corolario 3.8 La longitud de camino externo *lce* de un árbol-2 que tiene n nodos internos y cota inferior: $lce \geq (n + 1)\ \lg(n + 1)$.

Demostración Todo árbol-2 con n nodos internos tiene $(n + 1)$ nodos externos (véase el ejercicio 3.1). Aplicamos el lema 3.7. □

A menudo es conveniente que un procedimiento recursivo devuelva varias cantidades en una clase organizadora, aunque finalmente sólo se necesite una de esas cantidades. En este ejemplo no se pidió `numExt`, pero devolverlo después de las invocaciones recursivas simplificó considerablemente el resto del cálculo. Hay otro ejemplo en el ejercicio 3.13, que pide diseñar una función para calcular el peso máximo de un conjunto independiente de vértices de árbol.

3.5 Cómo demostrar que un procedimiento es correcto

Las cosas deben hacerse lo más sencillas que sea posible, pero no más.
—Albert Einstein

Casi todo mundo reconoce que demostrar que los programas son correctos es una tarea tan difícil que, *en general*, resulta prácticamente imposible. No obstante, demostrar la corrección puede ser una actividad valiosa para resolver problemas y producir programas que funcionen correctamente. El truco consiste en *programar adoptando un estilo con el cual resulte práctico demostrar la corrección*. Llamamos *estilo susceptible de demostración* a tal estilo. Queremos que las demostraciones nos ayuden, no que sean una carga adicional. Para que las demostraciones sean útiles, debemos escribir los algoritmos en un estilo susceptible de demostración, o al menos poder efectuar conversiones entre un estilo susceptible de demostración y un estilo eficiente, y viceversa.

En esta sección formaremos una colección de métodos de demostración, comenzando con construcciones simples que después se volverán más complejas, aunque sin dejar de ser manejables. El estilo que desarrollaremos se usará en los algoritmos de este libro. Sus cimientos son el paradigma de asignación única y la recursión. Presentaremos el paradigma de asignación única en la sección 3.5.3.

3.5.1 Definiciones y terminología

Un *bloque* es una sección de código que tiene un punto de entrada y un punto de salida. Los bloques son las subdivisiones principales del código de los programas y procedimientos. Un *procedimiento* es un bloque que tiene nombre, lo que permite invocarlo. Por lo regular, los procedimientos tienen *parámetros*, que para nuestros fines se designan como *de entrada* o *de salida*. Por sencillez, supondremos que ningún parámetro es al mismo tiempo de entrada y de salida; podemos designar dos parámetros para lograr los mismos efectos. Supondremos también que los parámetros de entrada *no se modifican* durante la ejecución del procedimiento. Podrían copiarse en datos locales (véase más adelante) si se desea modificarlos. Esta convención nos permite plantear las condiciones posteriores en términos de los parámetros de entrada sin tener que especificar que nos estamos refiriendo a los valores que tenían en el momento en que se entró en el procedimiento.

Una *función* es un procedimiento que tiene parámetros de salida; si hay varios parámetros de salida, podemos suponer que se juntan en un objeto de una *clase organizadora* (sección 1.2.2), y por tanto se pueden devolver con un solo enunciado `return`. Sólo hay un punto de salida, así que el enunciado `return` debe estar en ese punto. Este formalismo nos permite manejar las funciones como un caso especial de los procedimientos.

Un procedimiento a menudo hace referencia a *datos no locales*, que son un dato cualquiera que se haya definido fuera del encabezado y el cuerpo del procedimiento. Efectivamente, si un procedimiento no tiene parámetros de salida, los únicos efectos de su invocación serán efectos sobre datos no locales. Además, un procedimiento puede definir *datos locales*. Los parámetros del procedimiento también se pueden considerar como datos locales durante la ejecución del procedimiento.

Un bloque dentro de un procedimiento también puede hacer referencia a *datos no locales*, que son cualesquier datos que se hayan definido fuera del bloque. Es común llamar a éstos *datos globales*, y podrían ser datos de un bloque de nivel más alto, que encierra a éste, o datos externos al procedimiento, en cuyo caso serán válidas las reglas de visibilidad del lenguaje que se esté usando.

Vale la pena aclarar la forma en que se manejan los arreglos. Si un arreglo se pasa como parámetro, la *referencia* al arreglo se considera como dato local, mientras que el *contenido* del arreglo se considera como datos no locales. Así mismo, en Java, la *referencia* a un objeto es local, mientras que los *campos de ejemplar* del objeto son no locales. La actualización de datos no locales es muy importante para la eficiencia de algunos algoritmos, pero dificulta considerablemente demostrar su corrección.

3.5.2 Estructuras de control elementales

Las estructuras de control son mecanismos para hacer que se ejecuten diversos bloques. En un principio, sólo consideraremos tres estructuras de control (véase la figura 3.5): *sucesión* (bloque 1, luego bloque 2), *alternativa* (si *condición*, entonces bloque-se-cumple, si no, bloque-no-se-cumple) e *invocación de procedimiento*. La omisión de los ciclos **`for`** y **`while`** de nuestra metodología de demostración básica es intencional. Veremos adaptaciones de estas construcciones una vez que hayamos desarrollado la metodología básica (en la sección 3.5.4).

¿Podemos escribir algo que valga la pena sin ciclos? La respuesta, por sorprendente que parezca, es "sí". Empleando recursión, es posible, y a menudo más sencillo, escribir cualquier cálculo que se haya escrito originalmente con un ciclo.

"Demostrar corrección" significa demostrar ciertos enunciados lógicos acerca de un procedimiento. Al igual que una "garantía limitada", los enunciados se redactan con cautela, de modo que no sean tan generales que la demostración se vuelva excesivamente difícil. A continuación describiremos la forma que adoptan esos enunciados.

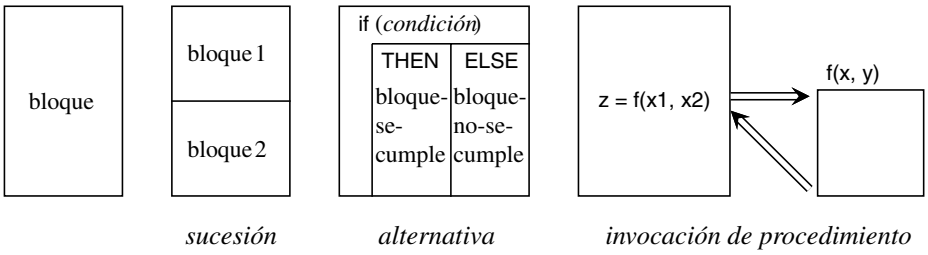


Figura 3.5 Estructuras de control de procedimientos elementales

Definición 3.4 Condición previa, condición posterior y especificación

Una *condición previa* es un enunciado lógico acerca de los parámetros de entrada y los datos no locales de un bloque (incluyendo procedimientos y funciones) que supuestamente se cumple en el momento en que se ingresa en el bloque. Una *condición posterior* es un enunciado lógico acerca de los parámetros de entrada, los parámetros de salida y los datos no locales de un bloque que supuestamente se cumple en el momento en que se sale del bloque. Las *especificaciones* de un bloque son las condiciones previas y posteriores que describen el comportamiento correcto del bloque. ■

Todo bloque (incluidos procedimientos y funciones) debe tener especificaciones si estamos tratando de demostrar corrección.

Para demostrar un comportamiento correcto, basta con demostrar un lema que tiene la forma siguiente.

Propuesta 3.9 (Forma general de lema de corrección) Si se cumplen todas las *condiciones previas* cuando se ingresa en el bloque, se cumplirán todas las *condiciones posteriores* cuando se salga del bloque. □

Supóngase que un bloque se subdivide mediante la construcción de *sucesión*: bloque 1, luego bloque 2. Para demostrar la corrección del bloque, basta con demostrar un lema que tiene esta forma:

Propuesta 3.10 (Forma de lema de corrección para sucesión)

1. Las condiciones previas del bloque implican las condiciones previas del bloque 1.
2. Las condiciones posteriores del bloque 1 implican las condiciones previas del bloque 2.
3. Las condiciones posteriores del bloque 2 implican las condiciones posteriores del bloque. □

Supóngase que un bloque se subdivide mediante la construcción de *alternativa*: si (*condición*), entonces bloque-se-cumple, si no, bloque-no-se-cumple. Para demostrar la corrección del bloque, basta con demostrar un lema que tiene la forma siguiente:

Propuesta 3.11 (Forma de lema de corrección para alternativa)

1. Las condiciones previas del bloque *y* el cumplimiento de la *condición* implican las condiciones previas del bloque-se-cumple.
2. Las condiciones posteriores del bloque-se-cumple *y* el cumplimiento de la *condición* (en el momento en que se ingresa en el bloque-se-cumple) implican las condiciones posteriores del bloque.
3. Las condiciones previas del bloque *y* el incumplimiento de la *condición* implican las condiciones previas del bloque-no-se-cumple.
4. Las condiciones posteriores del bloque-no-se-cumple *y* el incumplimiento de la *condición* (en el momento en que se ingresa en el bloque-no-se-cumple) implican las condiciones posteriores del bloque. □

La figura 3.6 muestra cómo se combinan las partes de cada lema de las propuestas 3.10 y 3.11 para producir una demostración de la forma de la propuesta 3.9.

Supóngase que un bloque consiste en una invocación de procedimiento. Para demostrar la corrección del bloque, basta con demostrar un lema que tiene esta forma:

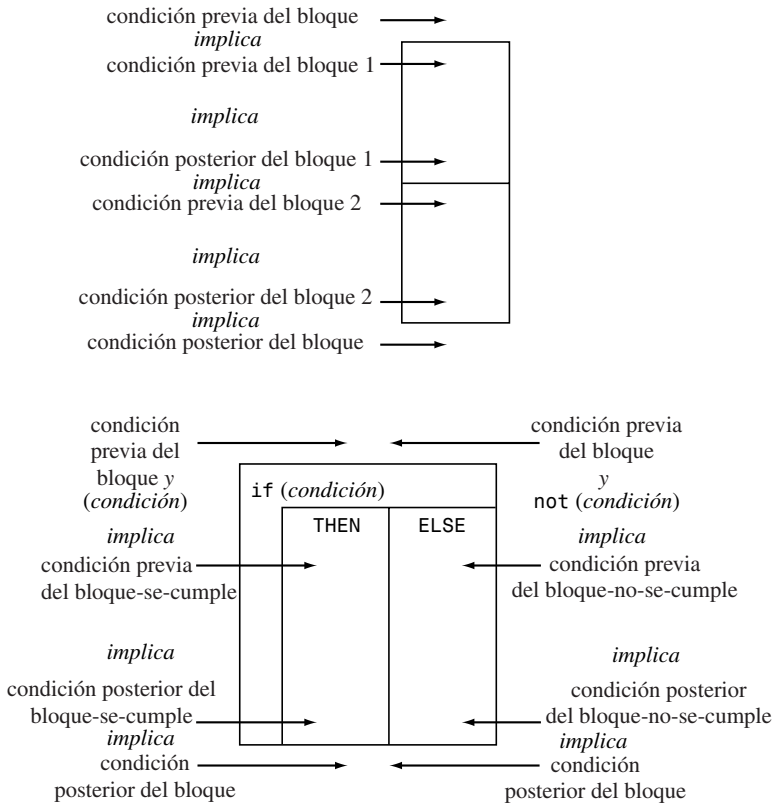


Figura 3.6 Cadenas de inferencias para demostrar que las condiciones previas del bloque implican sus condiciones posteriores, en los casos de *sucesión* y *alternativa*.

Propuesta 3.12 (Forma de lema de corrección para invocación de procedimiento)

1. Las condiciones previas del bloque implican las condiciones previas del procedimiento invocado con sus parámetros reales.
2. Las condiciones posteriores del procedimiento invocado con sus parámetros reales implican las condiciones posteriores del bloque. □

Es importante observar que *no* tenemos que demostrar la corrección del procedimiento invocado para demostrar la corrección del bloque que contiene la invocación; la corrección del procedimiento invocado es una cuestión aparte.

Hemos descrito la estructura de las demostraciones que nos permiten demostrar el comportamiento correcto de un bloque, pero no hemos entrado en pormenores de cómo demostrar hechos específicos acerca de enunciados específicos de un programa. Éste es un tema altamente técnico y complejo.

Por ejemplo, supóngase que vemos un enunciado en Java, "`x=y+1`". ¿Qué enunciado lógico sabemos que se cumple después de ese enunciado (es decir, cuál es la condición posterior del enunciado)? Es tentador decir que la condición posterior es la ecuación $x = y + 1$. Pero, ¿y si suponemos que el enunciado es "`y=y+1;`"?, ¿o que tenemos la *sucesión* de enunciados "`x=y+1; y=z;`"?

En la práctica, nos apoyamos en argumentos de "sentido común" en lugar de métodos de demostración formales. En lugar de tratar de deducir qué enunciados lógicos implica el código del procedimiento, nos concentramos en si se logran o no las condiciones posteriores deseadas y tratamos de idear argumentos *ad hoc* para sacar esa conclusión. El tema que sigue describe un mejor enfoque.

3.5.3 El paradigma de asignación única

Durante las primeras investigaciones de los estilos de programación susceptibles de demostración, se identificaron dos construcciones que dificultaban mucho demostrar la corrección: el enunciado *ir a* (*go to*) y el enunciado de *asignación*. Se consideró poco práctico eliminar los enunciados de asignación, así que los investigadores se abocaron a hacer innecesario el uso del enunciado *ir a*, desarrollando así el campo de la programación estructurada. Lamentablemente, incluso eliminando los enunciados *ir a*, las demostraciones suelen ser demasiado complejas como para que resulten prácticas.

En fechas más recientes, se ha vuelto a examinar la cuestión de eliminar los enunciados de asignación. La metodología emergente consiste en eliminar las asignaciones que *sobreescriben*. Es decir, una vez que se crea una variable, sólo puede recibir una asignación; el valor asignado no puede modificarse (sobreescribirse) posteriormente. Puesto que el valor de una variable no puede cambiar mientras existe, el razonamiento en torno a la variable se simplifica mucho. Éste es el *paradigma de asignación única*.

Se han creado varios lenguajes de programación que incorporan la restricción de una sola asignación, como Prolog, ML, Haskell, Sisal (acrónimo de "flujos e iteración en un lenguaje de asignación única") y SAC (acrónimo de "C con asignación única").

También se ha demostrado que los programas escritos en otros lenguajes, incluidos C, Fortran y Java, se pueden convertir a una forma de asignación única sin alterar el cálculo que efectúan. Tales transformaciones se utilizan en la optimización de compiladores y la detección de código paralelizable. Se ha descubierto que los programas se pueden analizar mucho más a fondo si primero se convierten a la forma de asignación única. (Véanse las Notas y Referencias al final del capítulo.) ¿Podemos aprovechar el paradigma de asignación única en la programación cotidiana?

El *paradigma de asignación única* no se puede aplicar universalmente, pero sí puede aplicarse con gran facilidad a variables locales de código que no tiene ciclos. El código sin ciclos incluye código con invocaciones de procedimiento recursivas, por lo que esta limitación no es tan severa que haga inútil al paradigma. De hecho, el compilador de Sisal transforma tras bambalinas los ciclos **`for`** y **`while`** en invocaciones de procedimiento recursivas para poder utilizar el paradigma de asignación única en el programa transformado. Luego, ya vigente el paradigma de asignación única, el compilador de Sisal puede deducir automáticamente cuáles secciones del código se pueden ejecutar en paralelo. No obstante, es posible usar una forma limitada de la asignación única con ciclos **`for`** y **`while`**.

Recordemos los enunciados de asignación que dificultaron nuestro razonamiento al principio de la sección. Dentro del paradigma de asignación única, "`x=y+1;`" *sí* implica la ecuación $x = y + 1$ en todo momento en que x tiene un valor. Los enunciados problemáticos "`y=y+1;`" y "`x=y+1; y=z;`" violan el paradigma porque asignan un valor a y por segunda vez.

En un procedimiento sin ciclos en el que *x* y *y* son variables locales, siempre podremos efectuar el cálculo deseado definiendo unas cuantas variables locales adicionales.

Ejemplo 3.5

Para arreglar el enunciado "`y=y+1;`" escribimos "`y1=y+1;`" en su lugar y obtenemos la *ecuación* válida $y1 = y + 1$. Para arreglar el enunciado "`x=y+1; y=z;`" escribimos "`x=y+1; y1=z;`" y obtenemos dos ecuaciones válidas, $x = y + 1$ y $y1 = z$. En ambos casos, todas las referencias posteriores a `y` en esta rama del procedimiento se harán a `y1`, de modo que accedan al valor modificado. ■

Examinemos otra dificultad común: una variable sólo se actualiza en una rama de una alternativa, pero se usa después de que las dos ramas se han vuelto a fusionar.

Ejemplo 3.6

Consideremos el fragmento de código:

```
1.   if(y < 0)
2.       y = 0;
3.   x = 2 * y;
```

Según lo que dijimos antes, deberíamos definir una nueva variable local `y1` y sustituir la línea 2 por "`y1 = 0`". Pero, ¿y la línea 3? Al parecer no sabríamos si usar `y` o `y1`. La solución es obedecer la regla de que si a una variable local se le asigna un valor en una rama de una alternativa, se le debe asignar un valor apropiado en todas las ramas. En este caso aparecerán múltiples enunciados de asignación en el código pero sólo uno de ellos podrá ejecutarse en cualquier invocación del procedimiento. El código modificado que se ajusta al paradigma de asignación única es

```
1.   if(y < 0)
2.       y1 = 0;
3.   else
4.       y1 = y;
5.   x = 2 * y1;
```

Ahora tenemos las siguientes relaciones lógicas muy claras entre las variables en cuestión (recordando que "$\Rightarrow$" es "implica" y "$\wedge$" es conjunción):

$$(y < 0 \Rightarrow y1 = 0) \wedge (y \geq 0 \Rightarrow y1 = y) \wedge (x = 2\ y1).$$

Los fanáticos de la eficiencia tal vez se horroricen ante la idea de crear variables extra, pero en realidad un compilador optimador puede determinar fácilmente si ya no se volverá a hacer referencia a la `y` original y usará su espacio para `y1`. ■

Sin embargo, cabe recordar que el paradigma de asignación única, si bien es muy útil para manejar variables locales, no es tan práctico al programar con arreglos. Es muy común que sea necesario actualizar los elementos de un arreglo, y obviamente no podemos darnos el lujo de definir todo un arreglo nuevo cada vez que actualicemos un elemento. Incluso si lo hiciéramos, tendríamos dificultades al tratar de deducir cualquier enunciado lógico que describiera el estado del arreglo. Los mismos problemas se presentan con objetos que tienen campos de ejemplar que es preciso actualizar.

Conversión en un procedimiento sin ciclos

Si queremos aplicar las herramientas de razonamiento de esta sección a un procedimiento que tiene un ciclo **for** o **while**, y el procedimiento es razonablemente compacto, el método más fácil podría ser convertir el ciclo en un procedimiento recursivo.

Ejemplo 3.7

En el algoritmo 1.1 se dio un procedimiento iterativo de Búsqueda Secuencial. El código que sigue presenta la versión recursiva y utiliza el paradigma de asignación única. Se aplican las propuestas 3.9 a 3.12 para demostrar su corrección. (No demostramos la corrección del algoritmo 1.1 debido a las complicaciones que surgen por el hecho de que las variables tienen múltiples valores.)

Recuerde el patrón de las *rutinas de búsqueda generalizadas* (vea la definición 1.12): si no hay más datos, fracasar; en caso contrario, examinar un dato; si es lo que estamos buscando, tener éxito; en caso contrario, buscar en los datos restantes. Es evidente que el procedimiento que sigue se ajusta a ese patrón.

Algoritmo 3.1 Búsqueda Secuencial, recursivo

Entradas: E, m, `num`, K, donde E es un arreglo de `num` elementos (indexados $0, \ldots,$ `num`-1), K es el elemento buscado, y $m \geq 0$ es el menor índice del segmento de arreglo en el que se buscará. Por sencillez, suponemos que K y los elementos de E son enteros, lo mismo que `num`.

Salidas: `respuesta`, una posición de K en E, dentro del intervalo $m \leq$ `respuesta` $\leq$ `num`, o -1 si K no se halla en ese intervalo.

Comentario: La invocación de nivel más alto deberá ser `respuesta = busquedaSecRec(E, 0, num, K)`.

```
int busquedaSecRec(int[] E, int m, int num, int K)
    int respuesta;
1.  if (m ≥ num)
2.          respuesta = −1;
3.  else if (E[m] == K)
4.          respuesta = m;
5.  else
6.          respuesta = busquedaSecRec(E, m+1, num, K);
7.  return respuesta;
```

Observe que `respuesta` aparece en tres enunciados de asignación, pero todos están en diferentes ramas del código, de modo que se respeta el paradigma de asignación única. Veamos qué implica la aplicación de las propuestas para verificar que el procedimiento es correcto.

Primero, necesitamos formular las condiciones previas de `busquedaSecRec`:

1. $m \geq 0$.
2. Para $m \leq i <$ `num`, `E[i]` está inicializado.

Ahora planteamos el objetivo, o condición posterior, que se deberá cumplir en la línea 7.

1. Si `respuesta` $= -1$, entonces para $m \leq i <$ `num`, `E[i]` $\neq K$.
2. Si `respuesta` $\neq -1$, entonces $m \leq$ `respuesta` $<$ `num` y `E[respuesta]` $= K$.

Ahora, según la propuesta 3.9, demostramos que si se cumplen las condiciones previas cuando se ingresa en `busquedaSecRec`, se cumplirán las condiciones posteriores cuando haya terminado. Vemos que el procedimiento se divide en tres casos alternos, líneas 2, 4 y 6, que convergen en la línea 7, el enunciado **`return`**. Aplicamos la propuesta 3.11 a cada alternativa. En cada caso, las condiciones que llevaron a esa alternativa son hechos adicionales que pueden servir para demostrar que se cumple la condición posterior para esa alternativa.

Si se llega a la línea 2, es porque se cumple la condición de la línea 1 ($m \geq$ `num`). Después de la línea 2, `respuesta` $= -1$, el flujo pasa a la línea 7. En este punto, el cumplimiento de la condición de la línea 1 implica que no hay índices en el intervalo $m \leq i <$ `num`, de modo que se cumple la condición posterior 1. La condición posterior 2 se cumple porque su hipótesis es falsa (recordemos la sección 1.3.3).

De forma similar, si se llega a la línea 4, es porque no se cumple la condición de la línea 1 (así que $m <$ `num`) y se cumple la condición de la línea 3 (`E[m]` $= K$). La misma línea 4 establece la ecuación (`respuesta` $= m$). La combinación de estos hechos hace que se cumpla la condición posterior 2. La ecuación (`respuesta` $= m$) y la condición previa 1 implican que la hipótesis de la condición posterior 1 es falsa; por tanto, se cumple la condición posterior 1.

Por último, si se llega a la línea 6, es porque no se cumplen las condiciones de las líneas 1 y 3 (así que $m <$ `num` y `E[m]` $\neq K$). Primero, necesitamos demostrar que "tenemos derecho a invocar" `busquedaSecRec` con los parámetros reales que se usan en la línea 6. Es decir, necesitamos verificar que se cumplan las condiciones previas de `busquedaSecRec` al igualarse con estos parámetros reales:

1. Si $m \geq 0$, entonces $m + 1 \geq 0$.
2. El intervalo $m + 1, \ldots,$ `num`-1 está contenido dentro de $m, \ldots,$ `num`-1, de modo que `E[i]` sí está inicializado ahí.

Ahora, por la propuesta 3.12, *podemos concluir que la invocación de procedimiento de la línea 6 cumple con sus condiciones posteriores*. Puesto que el valor de `respuesta` que se asigna en la línea 6 no es sino el valor devuelto por la invocación de la línea 6, satisface las condiciones posteriores de dicha invocación (con parámetro real $m + 1$). Estas condiciones posteriores y el enunciado `E[m]` $\neq K$ implican las condiciones posteriores de la invocación actual (con parámetro real m). Por ejemplo, si se devuelve -1, ello implica que K no está en `E[m+1]`$, \ldots,$ `E[num-1]`, y por tanto K no está en `E[m]`$, \ldots,$ `E[num-1]`. En caso contrario, `respuesta` $\geq m + 1$, de modo que `respuesta` $\geq m$ también.

Así pues, hemos demostrado que, siempre que se llega a la línea 7, se cumplen las condiciones posteriores requeridas. Sólo queda la duda de si es posible que nunca se llegue a la línea 7 debido a una recursión infinita. En la sección 3.5.6 abordamos esta cuestión. ■

El ejercicio 3.6 pide demostrar la corrección del algoritmo de Euclides para hallar el máximo común divisor de dos enteros, empleando las técnicas de esta sección.

3.5.4 Procedimientos con ciclos

Las propuestas 3.9 a 3.12 nos proporcionan un marco dentro del cual demostrar corrección en ausencia de ciclos **`for`** y **`while`**. Dentro de los ciclos suele ser imposible la asignación única, por lo que se hace necesario definir nombres de variable indexados *tanto* por el número de línea del procedimiento *como* por el número de pasadas por el ciclo para seguir la pista a todos los valores que adopta la misma variable de programa. Luego es necesario rastrear minuciosamente la historia de cada cambio de valor. En lugar de tratar de formalizar y llevar a cabo este procedimiento, cree-

mos que es más fácil en la práctica transformar el ciclo en un procedimiento recursivo, que requiere herramientas de demostración mucho más sencillas. En esta sección describiremos una forma relativamente mecánica de hacerlo.

De hecho, una vez que entendemos la relación entre el ciclo y la versión recursiva, suele ser innecesario efectuar realmente la transformación. Como paso de procesamiento previo, conviene hacer lo siguiente:

1. Declarar las variables locales dentro del cuerpo del ciclo, en la medida de lo posible, y ajustarse al paradigma de asignación única en esos casos. Es decir, dar a la variable únicamente un valor en cualquier pasada individual.
2. En el caso de variables que deben actualizarse (y que por fuerza se declaran fuera del ciclo), efectuar todas las actualizaciones al final del cuerpo del ciclo.

Estas reglas reducen al mínimo el número de casos distintos que es preciso considerar.

Las reglas generales para replantear un ciclo **`while`** con recursión son:

1. Las variables actualizadas en el ciclo se convierten en parámetros de entrada de procedimiento. Sus valores iniciales en el momento en que se ingresa en el ciclo corresponden a los parámetros reales en la invocación de nivel más alto del procedimiento recursivo. Llamamos a éstos *parámetros activos*.
2. Las variables a las que se hace referencia en el ciclo pero que se definieron antes y no se actualizan en el ciclo también se convertirán seguramente en parámetros, porque de otra manera no estarían accesibles en el nuevo procedimiento recursivo. Sin embargo, estos parámetros simplemente se "pasan a través", de invocación en invocación, así que los llamamos *parámetros pasivos*. Para los fines del análisis (si no vamos a convertir realmente el código), podemos tratar los parámetros pasivos como variables globales.
3. Lo primero que hace el procedimiento recursivo es simular la condición del **`while`** y regresar (es decir, saltar al enunciado **`return`** del nuevo procedimiento recursivo) si dicha condición no se cumple.
4. Un enunciado **`break`** también corresponde a un retorno del procedimiento.
5. Si se llega al final del cuerpo del **`while`**, se efectúa una invocación recursiva. Los parámetros reales de la invocación son los valores *actualizados* de las variables empleadas en el cuerpo del ciclo, las cuales estarán concentradas al final del cuerpo del ciclo si efectuamos el procesamiento previo sugerido.

Las reglas para los ciclos **`for`** son similares.

Esta transformación se ilustra con la función factorial en la figura 3.7. Observe que *n* es un parámetro pasivo.

Con excepción de la línea 7, el cuerpo del ciclo de `cicloFact` sigue el paradigma de asignación única. Así, podemos analizar el cuerpo del ciclo utilizando las propuestas 3.9 a 3.12 y empleando ecuaciones entre las variables, sin tener que efectuar la complicada indización o rotulación que suele ser necesaria cuando una variable adopta muchos valores distintos durante la ejecución del procedimiento. Por lo menos, podemos hacer esto hasta la línea 7, donde los valores de las variables cambian como preparación para la siguiente pasada. Además, si logramos visualizar este cambio como una nueva invocación de procedimiento con nuevos parámetros reales *y un problema de menor tamaño*, podremos tratar de demostrar algo acerca de él utilizando inducción. En este sentido limitado, podemos usar el paradigma de asignación única en procedimientos con ciclos.

```
int cicloFact(int n)
    int k, f;
1.  k = 1;
2.  f = 1;
3.  while (k ≤ n)
4.     {
5.     int fnueva = f * k;
6.     int knueva = k + 1;
7.     k = knueva; f = fnueva;
8.     }
9.  return f;
```

```
int fact(int n)
9.  return factRec(n, 1, 1);

int factRec(int n, int k, int f)
    int resp;
3a. if (k > n)
3b.     resp = f;
4.  else
5.     int fnueva = f * k;
6.     int knueva = k + 1;
7.     resp = factRec(n, knueva,
       fnueva);
   return resp;
```

Figura 3.7 Transformación de un ciclo **while** en una función recursiva. Se han omitido las llaves no relacionadas con la transformación.

3.5.5 Demostraciones de corrección como herramienta de depuración

Una de las grandes virtudes prácticas de las demostraciones de corrección —incluso las demostraciones "mentales" muy informales— es que a menudo señalan errores en los procedimientos incluso antes que se inicie la codificación y las pruebas. En parte esto se debe a la simple disciplina de pensar en las condiciones previas y posteriores del procedimiento y *escribirlas como comentarios en el código*. (Incluso si la demostración será "mental", no debe omitirse este paso de documentación.)

Muchos errores de programa son simples y son obvias las diferencias entre las condiciones previas de un procedimiento y las condiciones reales que prevalecen en el momento en que se invoca. Muchos otros se deben a las correspondientes diferencias respecto a las condiciones posteriores. Estas faltas de concordancia suelen hacerse evidentes tan pronto como se considera la propuesta 3.12.

Si el problema es menos obvio, y todo "se ve bien", es conveniente tratar de construir la demostración empleando las propuestas para pasar de bloque en bloque en trozos de tamaño razonable. Esto implica preguntar, para cada trozo que se esté tratando como un bloque, "¿Qué se supone que va a lograr este trozo?" y luego "¿Qué necesita cumplirse para que lo logre?" Ahora bien, ¿los trozos anteriores hacen que se cumpla eso?

Si hay un error en el código, y el análisis es cuidadoso, *el punto en el que falle la demostración nos indicará dónde está el error*. Es decir, el error seguramente se encontrará en alguno de los dos bloques que están a ambos lados de la frontera en la que no concuerdan las condiciones posteriores y las previas.

Por ejemplo, en la búsqueda secuencial recursiva (algoritmo 3.1), si la condición de la línea 1 se escribiera erróneamente (`m ≥ num−1`), no se implicaría la condición posterior 1 después de la línea 2, y se habría localizado el error.

En otro ejemplo, supóngase que se modifica el algoritmo 3.1 intercambiando las líneas 1-2 y las líneas 3-4, es decir, la línea 1 es ahora "`if (E[m] == K)`". Todos los enunciados mencionados en la demostración que dimos se pueden repetir cambiando el número de línea, pero el procedimiento tiene un error. Para detectar ese error durante una verificación, tenemos que percatar-

nos de que una condición previa de *cualquier* enunciado que evalúa una expresión es que todos los elementos de datos de la expresión han recibido valores, es decir, que no estamos accediendo a variables o campos de ejemplar no inicializados. Si *m* podría ser mayor que `num-1`, no tendremos esa certeza. Una vez más, el intento de demostración revela el error, pero sólo si estamos efectuando un examen muy cuidadoso. Siempre es recomendable preguntar, al revisar el código, "¿Ya se inicializó este elemento de datos?"

3.5.6 Terminación de procedimientos recursivos

Si hay procedimientos recursivos (las propuestas 3.9 a 3.12) los lemas descritos en la sección 3.5.2 demuestran lo que se conoce como corrección *parcial*, porque no se ocupan de averiguar si el procedimiento termina o no. Para completar la demostración de corrección *total*, es necesario demostrar que cada invocación de procedimiento recursivo está operando sobre un problema más pequeño que el problema que está resolviendo el procedimiento invocador.

En el punto en el que es necesario demostrar que se satisfacen las condiciones previas de la invocación recursiva, también se plantea que la estructura o el "tamaño de problema" que se está pasando a la invocación recursiva es menor que el del procedimiento invocador. Al igual que en otros aspectos de la corrección, en la práctica se usan argumentos de qué tan razonable es algo, los cuales recurren al sentido común, en lugar de demostraciones formales con axiomas y reglas de inferencia.

En muchos casos, el tamaño del problema es un entero no negativo, como el número de elementos de un subintervalo, el número de elementos de una lista ligada, etc. Por ejemplo, en el algoritmo 3.1 de la sección 3.5.2, es útil definir el "tamaño del problema" como $n = (\texttt{num} - m)$, el número de elementos que no se han examinado. Esta diferencia se decrementa en uno desde la invocación actual a la invocación recursiva, y la recursión termina si la diferencia llega a cero.

En algunos casos, es posible usar directamente un orden parcial definido para la estructura que se está pasando como parámetro de entrada, digamos el orden parcial *subárbol* (véase la figura 3.3). Por ejemplo, en un procedimiento para recorrer un árbol binario, como el de la figura 3.4 si el parámetro de entrada del procedimiento es el árbol *T*, y *T* no es un caso base, cada subárbol de *T* será "menor que" *T* dentro de este orden parcial. Por tanto, el procedimiento recursivo terminará con cualquier estructura de árbol binario que esté formada correctamente.

Hablando con precisión técnica, un procedimiento que efectúe recursión con un árbol binario deberá tener como condición previa que su parámetro de entrada *T* sea una estructura de árbol binario correctamente formada; en particular, no debe tener ciclos. Una razón por la que especificamos el tipo de datos abstracto Árbol Binario de forma no destructiva (en la sección 2.3.3) es que así se cumple automáticamente esta condición.

3.5.7 Corrección de Búsqueda Binaria

A continuación demostraremos la corrección del procedimiento recursivo `busquedaBinaria` con cierto detalle (véase el algoritmo 1.4, Búsqueda Binaria). Esto servirá para ilustrar el uso de la inducción para demostrar la corrección de un procedimiento recursivo. Una demostración por inducción establece la corrección total de un procedimiento recursivo sin ciclos; es decir, establece que el procedimiento termina, además de establecer que sus condiciones previas implican sus condiciones posteriores. (Si el procedimiento recursivo invoca subrutinas, entonces la corrección de las subrutinas se añade como *hipótesis* al teorema de corrección del procedimiento recursivo que se está demostrando.)

```
     int busquedaBinaria(int[] E, int primero, int ultimo, int K)
1.       if (ultimo < primero)
2.           indice = −1;
3.       else
4.           int medio = (primero + ultimo)/2;
5.           if (K == E[medio])
6.                 indice = medio;
7.           else if (K < E[medio])
8.                 indice = busqedaBinaria(E, primero, medio−1, K);
9.           else
10.                indice = busquedaBinaria(E, medio+1, ultimo, K);
11.      return indice;
```

Figura 3.8 Procedimiento `busquedaBinaria`, repetido del algoritmo 1.4.

Definimos el tamaño del problema de `busquedaBinaria` como $n =$ `ultimo` $-$ `primero` $+ 1$, el número de elementos del intervalo de E en el que se buscará. Repetimos el procedimiento en la figura 3.8 para comodidad del lector.

Lema 3.13 Para toda $n \geq 0$, si se invoca `busquedaBinaria(E, primero, ultimo, K)` y el tamaño del problema es (`ultimo` $-$ `primero` $+ 1$) $= n$, y `E[primero]`, ..., `E[ultimo]` están en orden no decreciente, se devolverá -1 si K no está en E dentro del intervalo `primero`, ..., `ultimo` y se devolverá `indice` tal que $K = E$[`indice`] en caso contrario.

Demostración La demostración es por inducción con n, el tamaño del problema. El caso base es $n = 0$. En este caso, la condición de la línea 1 se cumple, se llega a la línea 2, y se devuelve -1.

Para $n > 0$, suponemos que `busquedaBinaria(E,` f, ℓ, `K)` satisface el lema para problemas de tamaño k tal que $0 \leq k < n$, y f y ℓ son cualesquier índices tales que $k = \ell - f + 1$. Dado que $n > 0$, la condición de la línea 1 no se cumple, `primero` $\leq$ `ultimo`, el control llega a la línea 4 y luego a la 5. Por la desigualdad anterior y la ecuación `medio` $= \lfloor$(`primero` $+$ `ultimo`)/2$\rfloor$, vemos que `primero` $\leq$ `medio` $\leq$ `ultimo`. Por tanto, `medio` está dentro del intervalo de búsqueda. Si la línea 5 da `true`, el procedimiento logrará su objetivo en la línea 6.

Para el resto de la demostración, suponemos que la línea 5 da `false`. Por las dos desigualdades anteriores y la definición de n, tenemos (por la transitividad de $\leq$):

$$(\texttt{medio} - 1) - \texttt{primero} + 1 \leq (n - 1),$$
$$\texttt{ultimo} - (\texttt{medio} + 1) + 1 \leq (n - 1),$$

de modo que la hipótesis inductiva es válida para las dos invocaciones recursivas de las líneas 8 y 10.

Ahora bien, si la línea 7 da `true`, se ejecutará la línea 8. Es fácil verificar que las condición previas de `busquedaBinaria` se satisfacen con los parámetros reales de la línea 8 (sólo cambió el tercer parámetro, y disminuyó). Por tanto, podemos suponer que la invocación logra el objetivo de `busquedaBinaria`. Si la invocación de la línea 8 devuelve un índice positivo, se habrá resuelto el problema actual. Si esa invocación devuelve -1, querrá decir que K no está en E dentro del intervalo `primero`, ..., `medio` $- 1$. Sin embargo, el hecho de que se cumple la condición de la línea 7 implica que K no está en E dentro del intervalo `medio`, ..., `ultimo`, así que es correc-

to que la invocación de procedimiento actual devuelva -1. Si la línea 7 de `false`, se ejecutará la línea 10, y el argumento será similar. □

Un aspecto de la demostración que vale la pena destacar es que, antes de poder suponer (justificadamente) que las invocaciones de las líneas 8 y 10 logran sus objetivos, tuvimos que verificar que se cumplían las condiciones previas de las invocaciones. Dado que muchos errores de lógica se deben a la invocación de procedimientos sin cumplir con sus condiciones previas, este tipo de verificación puede descubrir muchos errores.

3.6 Ecuaciones de recurrencia

Una ecuación de recurrencia define una función sobre los números naturales, digamos $T(n)$, en términos de su propio valor con uno o más enteros menores que n. En otras palabras, $T(n)$ se define inductivamente. Al igual que con todas las inducciones, hay casos base que se definen aparte, y la ecuación de recurrencia sólo es válida para n mayor que los casos base. Aunque nuestro interés primordial son las ecuaciones de recurrencia para funciones que describen los recursos empleados por los algoritmos (generalmente el tiempo de ejecución, el número de comparaciones de claves, o las veces que se efectúa alguna otra operación importante), esto no es requisito para tener una ecuación de recurrencia. Muchas funciones matemáticas interesantes se pueden definir con ecuaciones de recurrencia, como los conocidos números de Fibonacci, ecuación (1.13).

Las ecuaciones de recurrencia surgen de forma muy natural cuando se desea expresar los recursos empleados por procedimientos recursivos. Los objetivos de esta sección son mostrar la forma de deducir tales ecuaciones de recurrencia a partir del código del procedimiento, y describir algunos patrones de algoritmos que se presentan con frecuencia. En la sección 3.7 exploraremos la forma de resolver algunas de las ecuaciones de recurrencia típicas que surgen de esta manera. Puesto que podrían medirse varios recursos distintos (tiempo, espacio, número de comparaciones de claves, etc.), usaremos el término general *costo* para referirnos a la cantidad que la ecuación de recurrencia describe o acota.

Primero necesitaremos especificar alguna forma de medir el tamaño del problema que el procedimiento recursivo está resolviendo: llamemos n a ese tamaño. El miembro izquierdo de la ecuación de recurrencia será $T(n)$. Para formar el miembro derecho de la ecuación será preciso estimar el costo de los diversos bloques del procedimiento en función de n. En muchos casos el costo de un bloque será constante. Podemos decir que todas las constantes son 1 si nos satisface una respuesta que está dentro de un margen constante.

En nuestra terminología, una *subrutina* es cualquier procedimiento que no es recursivo con el que estamos analizando; es decir, ninguna sucesión de invocaciones de la subrutina puede llevarnos de vuelta a este procedimiento. Las cantidades relacionadas con una subrutina por lo regular llevan el subíndice S. Las cantidades relacionadas con invocaciones recursivas normalmente llevan el subíndice R.

Es fácil combinar los costos de los bloques en un análisis en el peor de los casos si el procedimiento no tiene ciclos.

1. Para una *sucesión* de bloques, *sumamos* los costos individuales.
2. Para una *alternación* de bloques, en la que ninguna alternativa es un caso base, usamos el *máximo* de las alternativas.

3. Si un bloque contiene una invocación de subrutina, determinamos el tamaño de sus parámetros reales en función de n. Por sencillez, suponemos que sólo se necesitan parámetros de un tamaño, que llamamos $n_S(n)$. Necesitamos conocer la función de costo, digamos T_S, de la subrutina. Entonces, el costo de esta invocación será $T_S(n_S(n))$.
4. Si un bloque contiene una invocación de procedimiento recursiva, determinamos el tamaño de su parámetro real, en función de n, y lo llamamos $n_R(n)$. Entonces, el costo de esta invocación recursiva será $T(n_R(n))$. Ésta es la misma T que está en el miembro izquierdo de la ecuación de recurrencia.

Los términos que aparecen en el miembro derecho de la ecuación y que no contienen la función T (que aparece en el miembro izquierdo) son el *costo no recursivo* de la invocación de procedimiento. Usamos este nombre para distinguir este costo del costo total de la invocación de procedimiento, que incluye también los términos en T.

Para combinar los costos de los bloques en un análisis de caso promedio es necesario tratar la construcción de *alternativa* de forma distinta. Ponderamos el costo de cada alternativa con la probabilidad de que se presente, y sumamos los costos ponderados para dar el costo *esperado* o *promedio* de ese bloque. Además, si el tamaño de los subproblemas ($n_S(n)$ y $n_R(n)$) puede variar con diferentes entradas, será preciso promediar los costos de las invocaciones de subrutinas y las invocaciones recursivas. Por todo esto, el análisis del caso promedio suele ser mucho más difícil que el análisis en el peor de los casos.

Ejemplo 3.8

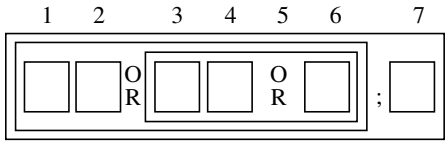
Como aplicación sencilla de estas reglas, consideremos la función recursiva

busquedaSecRec(E, m, num, K)

del algoritmo 3.1. Repetiremos aquí el cuerpo de su procedimiento, para comodidad del lector.

```
1.  if (m ≥ num)
2.       respuesta = −1;
3.  else if (E[m] == K)
4.       respuesta = m;
5.  else
6.       respuesta = busquedaSecRec(E, m+1, num, K);
7.  return respuesta;
```

Especificamos como medida del tamaño del problema el número de elementos del arreglo E que podrían contener la clave K que se busca. Entonces, $n =$ `num` $- m$, donde m y `num` son el segundo y el tercer parámetros reales de la invocación actual. Descompongamos el procedimiento en bloques para poder aplicar las reglas. Los bloques se pueden describir con su intervalo de números de línea. El procedimiento completo es 1-7, y se descompone como se sugiere en el diagrama siguiente, en el que "OR" denota *alternativas* y ";" denota *sucesión*.



El caso base es el bloque 2-2 y se excluye de la ecuación de recurrencia, que sólo es válida para los casos no base. Todos los bloques más internos son enunciados simples, con excepción de 6-6. Si el costo es tiempo de ejecución, suponemos que los enunciados sencillos tienen un costo constante y usamos 1 para representar cualquier constante (de modo que $1+1=1$ en este contexto). Si el costo es el número de alguna operación dada, contamos las operaciones. Supondremos, como caso específico, que el costo es el número de comparaciones con un elemento del arreglo. Así pues, la línea 3 tiene un costo de 1 y los demás enunciados simples son gratuitos en este modelo de costos.

Examinando la invocación de la línea 6 del algoritmo 3.1, vemos que el segundo y el tercer parámetros reales son $m + 1$ y `num`, de modo que el tamaño de su problema es `num` $-$ $(m + 1) = n - 1$. Por tanto, el costo de 6-6 es $T(n - 1)$. El costo de todo el bloque se obtiene a partir de los costos de los enunciados empleando máx para combinar alternativas y $+$ para combinar bloques sucesivos. El bloque 2-2 queda excluido como alternativa aquí. Observe que 1-7 es la suma de 1-6 y 7-7, y nos da la expresión del miembro derecho de la ecuación de recurrencia para `busqueda-SecRec`:

$$\big(0 + (1 + \text{máx}(0, T(n - 1)))\big) + 0.$$

Simplificando, vemos que la ecuación de recurrencia es $T(n) = T(n - 1) + 1$. El costo no recursivo es 1 en este caso.

Los casos base siempre son problemas pequeños, por lo que suponemos que siempre tienen costo unitario cuando el costo es tiempo. Sin embargo, aquí estamos contando comparaciones, así que $T(0) = 0$. ■

Ejemplo 3.9

Como ejemplo adicional, consideremos el procedimiento Búsqueda Binaria, algoritmo 1.4, que repetimos en la sección 3.5.7. El tamaño del problema es $n =$ `ultimo` $-$ `primero` $+ 1$. La medida de costo es, una vez más, las comparaciones de claves, así que la línea 5 cuesta 1. En las líneas 8 y 10 las invocaciones recursivas se efectúan con problemas de tamaño $n/2$ o $(n - 1)/2$, pero se trata de alternativas, así que el costo de la combinación es el máximo, no la suma. Ninguno de los demás enunciados efectúa comparaciones de claves, así que la ecuación de recurrencia es

$$T(n) = T(n/2) + 1.$$

En este procedimiento sólo se efectúa realmente una invocación recursiva, aunque aparecen dos. En el capítulo 4 veremos procedimientos para ordenar en los que sí se efectúan las dos invocaciones recursivas, y sus ecuaciones de recurrencia tienen términos en el miembro derecho para cada invocación recursiva. ■

Surgen problemas si no se conocen con mucha exactitud los tamaños $n_S(n)$ o $n_R(n)$. Por ejemplo, en un recorrido de árbol binario con n nodos (sección 2.3.3), sabemos que los subárboles izquierdo y derecho tienen en total $n - 1$ nodos, pero no sabemos cómo se divide esta suma entre los dos. Supóngase que introducimos una variable adicional r para representar el tamaño del subárbol derecho. Entonces, llegaremos a la ecuación de recurrencia

$$T(n) = T(n - 1 - r) + T(r) + 1, \qquad T(0) = 1.$$

Por fortuna, podemos determinar por sustitución que la función $T(n) = 2n + 1$ resuelve esta recurrencia sin necesidad de conocer el valor de r. En general no seremos tan afortunados, y el

comportamiento diferirá con distintos valores de r. Este problema tiene que resolverse en Quicksort (sección 4.4.3).

Ecuaciones de recurrencia comunes

Podemos describir varias categorías de ecuaciones de recurrencia que se presentan a menudo y que se pueden resolver (hasta cierto punto) empleando métodos estándar. En todos los casos, "subproblema" se refiere a un caso más pequeño del problema principal, que se resolverá con una invocación recursiva. Los símbolos b y c son constantes.

Divide y vencerás: En muchos casos del paradigma *divide y vencerás*, se sabe que el tamaño de los subproblemas es $n/2$ o alguna otra fracción fija de n, el tamaño del problema actual. Ejemplos de este comportamiento son Búsqueda Binaria (sección 1.6), que ya vimos, y algoritmos que estudiaremos en el capítulo 4: Mergesort (sección 4.6) y operaciones de montón (sección 4.8.3). Por ejemplo, en la sección 4.6 deduciremos esta ecuación de recurrencia para T_{MS}, el número de comparaciones efectuadas por Mergesort:

$$T_{MS}(n) = T_{MS}\left(\frac{n}{2}\right) + T_{MS}\left(\frac{n}{2}\right) + M(n), \qquad T_{MS}(1) = 0. \tag{3.2}$$

El costo $M(n)$ surge de una invocación de subrutina. Necesitaremos saber en qué consiste esa función para poder seguir resolviendo $T_{MS}(n)$.

En general, en los problemas del tipo divide y vencerás, el problema principal de tamaño n se puede dividir en b subproblemas ($b \geq 1$) de tamaño n/c ($c > 1$). También existe cierto costo no recursivo $f(n)$ (para dividir el problema en subproblemas y/o combinar las soluciones de los subproblemas y así resolver el problema principal).

$$T(n) = b\,T\left(\frac{n}{c}\right) + f(n). \tag{3.3}$$

Llamamos a *b factor de ramificación*.

Recorta y vencerás: El problema principal de tamaño n se puede "recortar" a un subproblema de tamaño $n - c$, donde $c > 0$, con costo no recursivo $f(n)$ (para crear el subproblema y/o extender la solución del subproblema a una solución del problema total).

$$T(n) = T(n - c) + f(n). \tag{3.4}$$

Recorta y serás vencido: El problema principal de tamaño n se puede "recortar" a b subproblemas ($b > 1$), cada uno de tamaño $n - c$, donde $c > 0$, con costo no recursivo $f(n)$ (para dividir el problema en subproblemas y/o combinar las soluciones de los subproblemas y así resolver el problema principal). Llamamos a *b factor de ramificación*.

$$T(n) = b\,T(n - c) + f(n). \tag{3.5}$$

Si los subproblemas son de distinto tamaño, pero todos están dentro de cierto intervalo constante $n - c_{máx}$ a $n - c_{mín}$, se pueden obtener cotas superiores e inferiores utilizado $c_{máx}$ y $c_{mín}$, respectivamente, en lugar de c en la ecuación. Este caso también se considera en el ejercicio 3.11.

En la sección que sigue examinaremos una estrategia metódica para analizar estas ecuaciones de recurrencia típicas.

3.7 Árboles de recursión

Los árboles de recursión son una herramienta para analizar el costo (tiempo de ejecución, número de comparaciones de claves o alguna otra medida) de procedimientos recursivos para los que hemos deducido ecuaciones de recurrencia. Primero mostraremos cómo desarrollar un árbol de recursión a partir de una ecuación de recurrencia, empleando un ejemplo; luego describiremos el procedimiento general. De ese procedimiento general podremos deducir varias soluciones generales (lema 3.14, lema 3.15, teorema 3.16, teorema 3.17, ecuaciones 3.12 y 3.13). Estas soluciones cubren muchas de las ecuaciones de recurrencia que surgen en la práctica durante el análisis de algoritmos, y sirven como guía aproximada incluso cuando las ecuaciones de recurrencia no están exactamente en una de las formas estándar. No es necesario entender todos los detalles técnicos de esta sección para poder aplicar las soluciones generales mencionadas.

Cada nodo del árbol de recursión tiene dos campos: *tamaño* y *costo no recursivo*. Representamos un nodo así:

T(*tamaño*)	*costo no rec.*

El campo de tamaño contiene el parámetro real de T para este nodo. Incluimos el nombre de recurrencia T para recordar que el campo de tamaño no es un costo.

Ejemplo 3.10 Árbol de recursión de divide y vencerás simple

Consideremos la ecuación de recurrencia:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n.$$

Éste es un caso especial de la forma de la ecuación (3.3), con $b = 2$ y $c = 2$. Se trata de una forma un tanto simplificada de la ecuación de recurrencia para Mergesort, cabe indicar que surge en muchas situaciones. Seguiremos los pasos para desarrollar el árbol de recursión correspondiente. El primer paso, que ayuda a evitar errores de sustitución, consiste en reescribir la ecuación con una variable auxiliar (existe una analogía con la variable auxiliar de una hipótesis de inducción). Llamamos a ésta nuestra *copia de trabajo* de la ecuación de recurrencia.

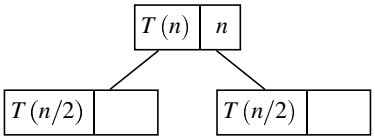
$$T(k) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n. \tag{3.6}$$

Se puede crear un nodo tan pronto como se conoce su campo de tamaño; después podremos usar ese campo para calcular un valor del campo de costo no recursivo. Estamos listos para crear el nodo raíz del árbol de recursión para $T(n)$; aquí, *tamaño* $= n$.

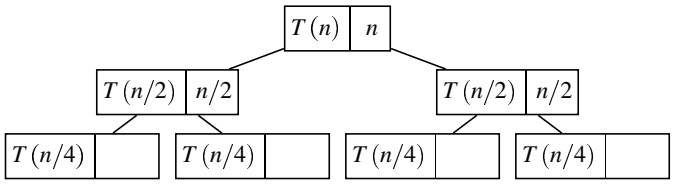
$T(n)$	

El proceso de determinar el campo de costo no recursivo y los hijos de un nodo incompleto se denomina *expansión* de ese nodo. Tomamos el campo de tamaño del nodo que vamos a expandir (en este caso n) y lo sustituimos por k en nuestra copia de trabajo, ecuación (3.6). Examinamos el miembro derecho resultante, que es $T(n/2) + T(n/2) + n$. Todos los términos con T se con-

vierten en hijos del nodo que estamos expandiendo, y todos los demás términos se convierten en el costo no recursivo de ese nodo, como sigue:



Puesto que todos los nodos que están a la misma profundidad tienen el mismo aspecto, los podemos generar en lotes. En general, cada nodo incompleto se debe generar según su propio campo de tamaño. Aquí todos los campos de tamaño son $n/2$, así que esta vez sustituimos $n/2$ por k en la ecuación (3.6), y vemos que el miembro derecho queda así: $T(n/4) + T(n/4) + n/2$. Por tanto, ahora tenemos



Continuamos así varios niveles hasta ver el patrón que el árbol está siguiendo. La figura 3.9 muestra el árbol después de expandirlo otro nivel; hay ocho hijos incompletos cuyos detalles no se muestran. Aquí vemos que a una profundidad d el parámetro de tamaño es $n/2^d$ y el costo no recursivo también resulta ser $n/2^d$. (Recordemos que la profundidad de la raíz es cero en nuestra convención.) En este sencillo ejemplo todos los nodos que están a la misma profundidad en el árbol son idénticos, pero no siempre sucede así. ■

Resumimos las reglas para desarrollar un árbol de recursión en la página siguiente.

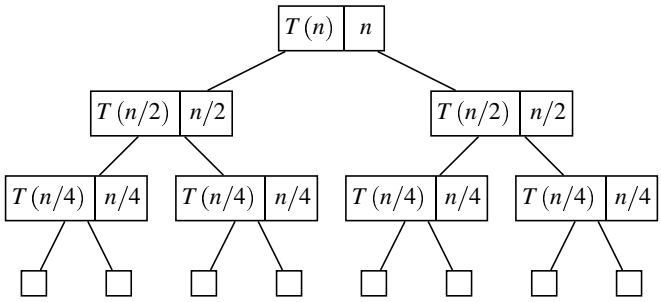


Figura 3.9 Tres niveles superiores de un árbol de recursión. No se muestran los campos de tamaño de los ocho hijos incompletos.

Definición 3.5 Reglas de árbol de recursión

1. La *copia de trabajo* de la ecuación de recurrencia usa una variable distinta de la que usa la copia original; se denomina *variable auxiliar*. Sea k la variable auxiliar para los fines de esta explicación. El miembro izquierdo de la copia original de la ecuación de recurrencia (supóngase que es $T(n)$) se convierte en el campo de tamaño del nodo raíz del árbol de recursión.
2. Un nodo incompleto tiene un valor en su campo de tamaño, pero no en su campo de costo no recursivo.
3. El proceso de determinar el campo de costo no recursivo y los hijos de un nodo incompleto se denomina *expansión* de ese nodo. Tomamos el campo de tamaño del nodo que vamos a expandir y lo sustituimos por la variable auxiliar k en nuestra copia de trabajo de la ecuación de recurrencia. Los términos resultantes que contienen a T en el miembro derecho de esa ecuación se convierten en hijos del nodo que se está expandiendo; todos los demás términos se convierten en el costo no recursivo de ese costo.
4. La expansión de un tamaño de caso base da un campo de costo no recursivo pero ningún hijo.

   Para simplificar la presentación, supondremos que la ecuación de recurrencia se definió de tal manera que ningún caso base tiene costo cero. Si la ecuación se presenta con casos base que cuestan cero, bastará con calcular los casos más pequeños cuyo costo no sea cero y usarlos como casos base en su lugar.

   De hecho, por lo regular supondremos que el caso base cuesta 1, para precisar los valores. Se pueden deducir variaciones si es necesario. ■

En cualquier subárbol del árbol de recursión, se cumple la ecuación siguiente:

$$\text{campo de tamaño de la raíz} = \sum \text{costos no recursivos de nodos expandidos} + \sum \text{campos de tamaño de nodos incompletos.} \qquad (3.7)$$

Es fácil demostrar esto por inducción. En el caso base, $T(n) = T(n)$. Después de una expansión, el nodo raíz se ha expandido y los hijos están incompletos, por lo que la ecuación (3.7) da exactamente la ecuación de recurrencia original, y así sucesivamente.

Ejemplo 3.11 Interpretación de un árbol de recursión

En el árbol de siete nodos del ejemplo 3.10 (con cuatro nodos incompletos), la ecuación (3.7) dice que $T(n) = n + 2(n/2) + 4\,T(n/4) = 2n + 4\,T(n/4)$. ■

La técnica para evaluar el árbol de recursión es la siguiente: primero se suman los costos no recursivos de todos los nodos que están a la misma profundidad; esto es la *suma de filas* para esa profundidad del árbol. Luego se suman todas las sumas de fila de todas las profundidades. Continuando el ejemplo de la figura 3.9, en la figura 3.10 se muestran algunas sumas de fila.

Para evaluar la sumatoria de sumas de fila es necesario (por lo regular) conocer la profundidad máxima del árbol de recursión. Ésta es la profundidad a la que el parámetro de tamaño se reduce a un caso base.

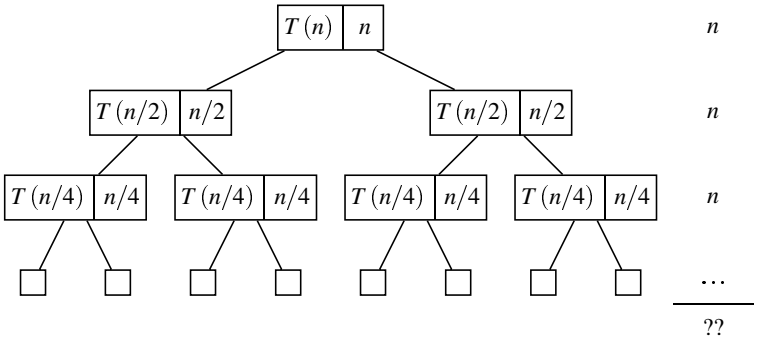


Figura 3.10 Sumatoria de costos no recursivos en un árbol de recursión. A la derecha se da la suma de fila de cada una de las tres primeras filas.

Ejemplo 3.12 Evaluación de un árbol de recursión

Para el árbol del ejemplo 3.10 (véase la figura 3.10), observamos que el tamaño en función de la profundidad del nodo d es $n/2^d$, así que los casos base se presentan aproximadamente a la profundidad $d = \lg(n)$. Puesto que cada una de las sumas de fila es n, el total para el árbol, que da el valor de $T(n)$, es aproximadamente $n \lg(n)$. ■

3.7.1 Divide y vencerás, caso general

Siguiendo los mismos pasos de los ejemplos 3.10 a 3.12, podemos evaluar la ecuación de recurrencia general de divide y vencerás [ecuación (3.3), que repetimos aquí], para obtener el orden asintótico de $T(n)$.

$$T(n) = b\,T\left(\frac{n}{c}\right) + f(n). \tag{3.8}$$

Esta sección se va a poner técnica, pero los lemas y teoremas se pueden entender y usar sin entender todos los pasos de las deducciones.

Primero, vemos que el parámetro de tamaño disminuye en un factor de c cada vez que se incrementa la profundidad (teníamos $c = 2$ en el ejemplo). Por tanto, los casos base (hojas del árbol) se presentan aproximadamente a la profundidad en que $(n/c^D) = 1$, donde D es la profundidad de los nodos de caso base. Despejamos D y obtenemos $D = \lg(n)/\lg(c) \in \Theta(\log(n))$. Sin embargo, no debemos apresurarnos a concluir que las sumas de fila son iguales en todas las profundidades.

Resulta útil saber *cuántas* hojas tiene el árbol. El factor de ramificación es b, así que el número de nodos a la profundidad D es $L = b^D$. Para expresar esto de forma más útil sacamos logaritmos: $\lg(L) = D \lg(b) = (\lg(b)/\lg(c)) \lg(n)$. El coeficiente de $\lg(n)$ es muy importante, así que le pondremos nombre.

Definición 3.6 Exponente crítico

Para b y c de la ecuación (3.3) (o la ecuación 3.8) definimos el *exponente crítico* como

$$E = \frac{\lg(b)}{\lg(c)}. \quad ■$$

Por el lema 1.1, parte 8, podemos usar cualquier base que nos convenga para los logaritmos de la fórmula de E, mientras sea la misma en el numerador y en el denominador. Con esta notación, el párrafo que precede a la definición ha demostrado que:

Lema 3.14 El número de hojas del árbol de recursión para la ecuación (3.8) es aproximadamente $L = n^E$, donde E es el exponente crítico según la definición 3.6. □

Suponiendo que el costo no recursivo es de 1 en las hojas, esto nos dice que el costo del árbol es *por lo menos* n^E. Incluso si los costos no recursivos de las hojas son cero, habrá costos distintos de cero en el nivel que está arriba de las hojas (o en algún número constante de niveles por arriba de las hojas, en un caso extremo). No obstante, sigue habiendo $\Theta(n^E)$ nodos en este nivel, por lo que sigue siendo vigente una cota inferior de $\Omega(n^E)$.

Resumamos lo que sabemos.

Lema 3.15 Con la notación de la explicación anterior, tenemos, aproximadamente:

1. El árbol de recursión tiene una profundidad de $D = \lg(n)/\lg(c)$, por lo que hay aproximadamente ese número de sumas de fila.
2. La suma de filas número cero es $f(n)$, el costo no recursivo de la raíz.
3. La D-ésima suma de fila es n^E, suponiendo que los casos bases cuestan 1, o por lo menos $\Theta(n^E)$.
4. El valor de $T(n)$, es decir, la solución de la ecuación (3.8), es la sumatoria de los costos no recursivos de todos los nodos del árbol, que es la sumatoria de las sumas de fila. □

En muchos casos prácticos, las sumas de fila forman una serie geométrica (o se pueden aproximar bien desde arriba y desde abajo con dos series geométricas). Recordemos que una serie geométrica tiene la forma $\sum_{d=0}^{D} ar^d$ (sección 1.3.2). La constante r se denomina *razón*. En la práctica se dan muchas simplificaciones basadas en el principio del teorema 1.13, parte 2, que dice que, para una serie geométrica cuya razón no es 1, la sumatoria está en Θ de su término más grande. Por este teorema y el lema 3.15, podemos concluir lo siguiente:

Teorema 3.16 (Teorema Maestro Pequeño) Con la notación de la explicación anterior y $T(n)$ definida por la ecuación (3.8):

1. Si las sumas de fila forman una serie geométrica creciente (a partir de la fila 0 en la parte más alta del árbol), entonces $T(n) \in \Theta(n^E)$, donde E es el *exponente crítico* definido en la definición 3.6. Es decir, el costo es proporcional al número de hojas del árbol de recursión.
2. Si las sumas de fila se mantienen aproximadamente constantes, $T(n) \in \Theta(f(n) \log(n))$.
3. Si las sumas de fila forman una serie geométrica decreciente, entonces $T(n) \in \Theta(f(n))$, que es proporcional al costo de la raíz.

Demostración En el caso 1 el último término domina la sumatoria. En el caso 2 hay $\Theta(\log(n))$ términos iguales. En el caso 3 el primer término domina la sumatoria. □

Si ahondamos técnicamente, podremos generalizar considerablemente este teorema. La generalización suele ser útil cuando la función $f(n)$ de la ecuación 3.8 implica logaritmos, porque

entonces es posible que las sumas de fila no tengan un comportamiento muy ordenado. (Se presenta una versión aún más general en el ejercicio 3.9.)

Teorema 3.17 (Teorema Maestro) Con la terminología de la explicación anterior, la solución de la ecuación de recurrencia

$$T(n) = b\,T\left(\frac{n}{c}\right) + f(n) \tag{3.9}$$

(replanteada a partir de las ecuaciones 3.3 y 3.8) tiene las formas de solución siguientes, donde $E = \lg(b)/\lg(c)$ es el *exponente crítico* que se indicó en la definición 3.6.

1. Si $f(n) \in O(n^{E-\epsilon})$ para cualquier ϵ positiva, entonces $T(n) \in \Theta(n^E)$, que es proporcional al número de hojas del árbol de recursión.
2. Si $f(n) \in \Theta(n^E)$, entonces $T(n) \in \Theta(f(n)\log(n))$, ya que la contribución de todos los nodos, sea cual sea su profundidad, es aproximadamente la misma.
3. Si $f(n) \in \Omega(n^{E+\epsilon})$ para cualquier ϵ positiva, y $f(n) \in O(n^{E+\delta})$ para alguna $\delta \geq \epsilon$, entonces $T(n) \in \Theta(f(n))$, que es proporcional al costo no recursivo en la raíz del árbol de recursión.

(Es posible que ninguno de estos casos sea el caso en cuestión.)

Demostración En la profundidad d hay b^d nodos y cada uno aporta un costo no recursivo de $f(n/c^d)$. Por tanto, tenemos la expresión general siguiente para la solución de la ecuación (3.8):

$$T(n) = \sum_{d=0}^{\lg(n)/\lg(c)} b^d f\left(\frac{n}{c^d}\right). \tag{3.10}$$

Nos limitaremos a bosquejar la demostración, que sigue un razonamiento similar al del teorema 3.16. (En las Notas y referencias hay fuentes que incluyen una demostración completa.) Consideremos el caso 3. Si hacemos caso omiso de los coeficientes, $f(n)$ es aproximadamente $n^{E+\epsilon}$, para alguna ϵ positiva. Entonces

$$f\left(\frac{n}{c^d}\right) \approx \frac{n^{E+\epsilon}}{(c^d)^{E+\epsilon}} \approx \frac{f(n)}{c^{Ed+\epsilon d}}.$$

Entonces $b^d\,f(n/c^d)$ es aproximadamente $f(n)b^d/(c^{Ed}c^{\epsilon d})$. Sin embargo, $c^E = b$ por identidades estándar, así que c^{Ed} en el denominador cancela b^d en el numerador. Al final tenemos $f(n)/c^{\epsilon d}$, que da una serie geométrica decreciente en d. El análisis de los demás casos es similar. □

3.7.2 Recorta y vencerás, o serás vencido

La situación de las ecuaciones (3.4) y (3.5) es diferente. Si el factor de ramificación es mayor que 1, tenemos la ecuación (3.5), que repetimos aquí para comodidad del lector:

$$T(n) = b\,T(n-c) + f(n). \tag{3.11}$$

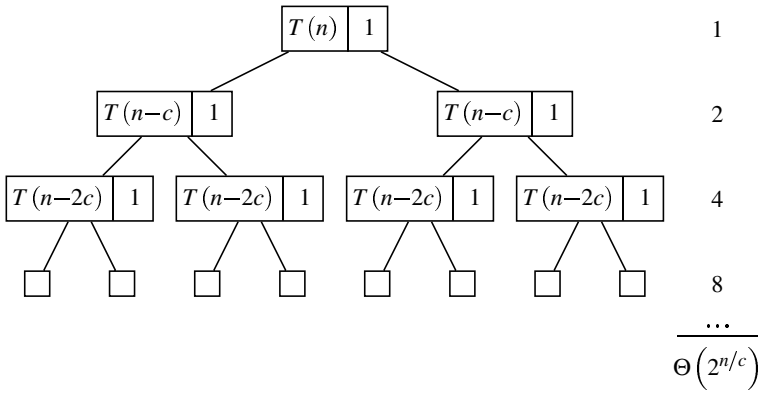


Figura 3.11 Sumatoria de costos no recursivos en un árbol de recursión de recorta y serás vencido

La figura 3.11 muestra el árbol de recursión para un ejemplo de la ecuación (3.11) con $f(k) = 1$. Puesto que el tamaño disminuye en c con cada incremento de 1 en la profundidad, los casos base están a aproximadamente $d = n/c$.

Como ilustra la figura 3.11, el total del árbol es exponencial en n, el tamaño del problema. Esto se cumple incluso con el supuesto, más favorable, de que $f(n) = 1$. La expresión general que sigue se puede obtener por inspección del árbol de recursión para la ecuación (3.11):

$$T(n) = \sum_{d=0}^{n/c} b^d f(n - cd) = b^{n/c} \sum_{h=0}^{n/c} \frac{f(c\,h)}{b^h} \qquad \text{(solución de la ecuación 3.11)} \qquad (3.12)$$

donde la segunda sumatoria usa $h = (n/c) - d$, de modo que h es cero en las hojas y aumenta hacia la raíz. En la mayor parte de los casos prácticos, la última sumatoria es $\Theta(1)$, lo que da $T(n) \in \Theta(b^{n/c})$. Esta función crece exponencialmente en n. En el ejercicio 3.11 se considera un caso más general de recorta y serás vencido. Como vimos en la sección 1.5, los algoritmos con tasa de crecimiento exponencial no podrán resolver los peores casos de problemas no pequeños.

No obstante, si el factor de ramificación b es 1 en la ecuación (3.11) (lo que da la ecuación 3.4), la expresión general de la ecuación (3.12) se vuelve mucho más amable:

$$T(n) = \sum_{d=0}^{n/c} f(n - cd) = \sum_{h=0}^{n/c} f(c\,h) \approx \frac{1}{c} \int_0^n f(x)\,dx \qquad \text{(solución de la ecuación 3.4)} \qquad (3.13)$$

Por ejemplo, si $f(n)$ es un polinomio n^α, entonces $T(n) \in \Theta(n^{\alpha+1})$. En cambio, si $f(n) = \log(n)$, entonces $T(n) \in \Theta(n \log(n))$. (Véase la sección 1.3.2.)

En síntesis, tenemos dos herramientas para evaluar el costo de un procedimiento recursivo: el árbol de recursión y la ecuación de recurrencia. Se trata de representaciones diferentes de la misma información. Se han desarrollado varias técnicas para evaluar formas de estos árboles y ecuaciones que se presentan con frecuencia. Incluso en una situación que no se ajusta a una for-

ma estándar, el árbol de recursión expresa la solución correcta de la ecuación de recurrencia; el único problema es que podría ser difícil de evaluar.

⋆ 3.7.3 Por qué funcionan los árboles de recursión

En esta sección explicaremos la relación entre el árbol de recursión de una ecuación de recurrencia dada y una función programada que calcula la solución por recursión. El lector puede omitir su lectura sin pérdida de continuidad.

Una forma de visualizar el árbol de recursión es imaginar que realmente programamos una función recursiva simple (llamémosla `evalT`(k)) para evaluar alguna ecuación de recurrencia, como las ecuaciones (3.2) a (3.5). El árbol de activación para la función programada corresponde con mucha exactitud al árbol de recursión. La ecuación de recurrencia tiene la forma $T(k) = f(k) + \ldots$ (términos con T). Suponemos que nuestra función recursiva `evalT` tiene un parámetro, k, que representa el tamaño del problema, y una variable local `costoNoRec` para guardar el valor calculado del costo no recursivo (es decir, $f(k)$). Olvidándonos del caso base, suponemos que el código de `evalT` es

```
costoNoRec = f(k);
return costoNoRec + ... (términos con evalT);
```

donde los términos con `evalT` son iguales a los términos con T del miembro derecho de la ecuación de recurrencia.

El *árbol de recursión* para esa ecuación de recurrencia, con $T(n)$ en la raíz, sería el *árbol de activación* de `evalT`(n). (Esto supone que es posible evaluar la función de costo no recursivo, $f(k)$, con enunciados simples.)

La revelación fundamental es que el valor que se devuelve al nivel más alto es exactamente la sumatoria de todos los valores de `costoNoRec` del árbol. Suponemos que la invocación de nivel más alto es `evalT`(n), para calcular el valor de $T(n)$ de la ecuación de recurrencia.

Ejercicios

Sección 3.2 Procedimientos recursivos

3.1 Demuestre que todo árbol-2 (definición 3.2) que tiene n nodos internos tiene $n + 1$ nodos externos.

3.2 En el lema 3.7 usamos el hecho de que $x \lg(x)$ es convexa. Demuéstrelo.

3.3 Demuestre que la longitud de camino externo *lce* de un árbol-2 que tiene m nodos externos satisface $lce \leq \frac{1}{2}(m2 + m - 2)$. Llegue a la conclusión de que $lce \leq \frac{1}{2}n(n + 3)$ para un árbol-2 que tiene n nodos internos.

3.4 La ecuación (1.13) definió la sucesión de Fibonacci como $F(n) = F(n - 1) + F(n - 2)$ para $n \geq 2$, $F(0) = 0$ y $F(1) = 1$. Demuestre (por inducción) el que sea correcto de los enunciados siguientes:

1. Para $n \geq 1$, $F(n) \leq 100\left(\frac{3}{2}\right)^n$.
2. Para $n \geq 1$, $F(n) \geq .01\left(\frac{3}{2}\right)^n$.

Se escogieron las constantes de modo que sea difícil adivinar cuál enunciado es correcto; el lector tendrá que basarse en su demostración.

Sección 3.5 Cómo demostrar que un procedimiento es correcto

3.5 Considere este procedimiento, que recibe dos arreglos como parámetros:

```
desplaInc(int[] A, int[] B)
     A[0] = B[0];
     B[0] ++;
     return;
```

Suponiendo que no hay desbordamiento de enteros, ¿se cumple forzosamente que $A[0] < B[0]$?

3.6 En este ejercicio se considera que todos los enteros son no negativos, por sencillez. Un *divisor* de un entero k es cualquier entero $d \neq 0$ tal que k/d no deja residuo. Un *divisor común* de un conjunto de enteros es un entero que es divisor de todos los enteros del conjunto. El algoritmo de Euclides para hallar el divisor común más grande (máximo común divisor, MCD) de dos enteros no negativos, m y n, se puede escribir sin usar división, así:

```
int mcd(int m, int n)
   int resp, nMenos;
1. if (m == 0)
2.      resp = n;
3. else if (m > n)
4.      resp = mcd(n, m)
5. else
6.      nMenos = n−m;
7.      resp = mcd(m, nMenos);
8. return resp;
```

Las condiciones previas de `mcd(m, n)` son que $m \geq 0$, $n \geq 0$ y por lo menos uno de m y n sea positivo. Se necesitarán algunas (no demasiadas) relaciones aritméticas para las demostraciones siguientes.

1. Si $a > b$, entonces $a - c > b - c$. [La ecuación (1.20) presenta otras variaciones.]
2. Si d es un divisor de k, entonces d es un divisor de $k - d$ y $k + d$. (Aunque hay que verificar por separado si $k - d < 0$.)
3. Si d es un divisor de k, entonces $d \leq k$ o $k = 0$.

Demuestre lo siguiente empleando inducción y los lemas de la sección 3.5.2, según sea necesario.

a. Si se satisfacen las condiciones previas de `mcd(m, n)`, entonces el valor devuelto por la función es *algún* divisor común de m y n.

⋆ **b.** Si se satisfacen las condiciones previas de `mcd(m, n)`, entonces el valor devuelto por la función es el *máximo* divisor común de m y n.

Sección 3.6 Ecuaciones de recurrencia

3.7 Suponga que se define la función M para todas las potencias de 2 y que está descrita por la ecuación de recurrencia y caso base siguientes:

$$M(n) = n - 1 + 2\,M(n/2)$$
$$M(1) = 0$$

a. Determine el orden asintótico de $M(n)$.
b. Obtenga una solución exacta de M cuando n es una potencia de 2.

3.8 Suponga que W satisface la ecuación de recurrencia y caso base siguientes (donde c es una constante):

$$W(n) = cn + W(\lfloor n/2 \rfloor)$$
$$W(1) = 1.$$

Determine el orden asintótico de $W(n)$.

★ **3.9** Otra estrategia para resolver ecuaciones de recurrencia de divide y vencerás implica cambios de variables y transformaciones de funciones. Éste es un ejercicio largo en el que se usan matemáticas un tanto complicadas; la última parte da una generalización del Teorema Maestro.

La ecuación inicial, al igual que en el teorema 3.17, es

$$T(n) = b\,T\left(\frac{n}{c}\right) = f(n).$$

Primero, nos limitaremos a n de la forma $n = c^k$ y supondremos que $T(1) = f(1)$. La variable k será un entero no negativo en todo este ejercicio. Realizaremos un cambio de variables definiendo $U(k) = T(c^k)$ para toda k. Luego efectuamos una transformación de funciones definiendo $V(k) = U(k)/b^k$ para toda k.

a. Deduzca la ecuación de recurrencia para $U(k)$ y determine el valor de $U(0)$. Se deberá eliminar totalmente la variable n.
b. Deduzca una ecuación de recurrencia para $V(k)$ y determine el valor de $V(0)$. El miembro izquierdo de la ecuación deberá ser $V(k)$ y el miembro derecho se deberá simplificar hasta donde sea razonable.
c. Replantee la ecuación de recurrencia para V como $V(i)$, introduciendo i como variable auxiliar. Luego exprese $V(k)$ como cierta sumatoria desde $i = 0$ hasta k.
d. Sea $E = \lg(b)/\lg(c)$, igual que en el teorema 3.17, para el resto del ejercicio. Demuestre que si $m = c^i$, entonces $m^E = b^i$.
e. Suponga que $f(m) \in \Theta(m^E)$. Observe que se trata del caso 2 del Teorema Maestro. (Se introdujo m como variable auxiliar para no confundirla con n, la cual queremos en la respuesta final.) Determine el orden asintótico de $V(k)$.
f. Convierta su expresión para $V(k)$ de la parte (e) en una expresión para $U(k)$, y luego para $T(n)$. (Deberá coincidir con el caso 2 del Teorema Maestro. La parte (d) puede ser de ayuda.)

g. Suponga ahora que $f(m) \in \Theta(m^E \log^a(m))$, donde E se define como en el teorema 3.17, para alguna constante positiva a. (Observe que $\log^a(m) = (\log m)^a$.) Determine el orden asintótico de $V(k)$.

h. Convierta su expresión para $V(k)$ de la parte (g) en una expresión para $U(k)$ y luego para $T(n)$. Llegue a la conclusión de que cuando $f(m) \in \Theta(m^E \log^a(m))$, la solución para la recurrencia de divide y vencerás es

$$T(n) \in \Theta\left(n^E \log^{a+1}(n)\right). \tag{3.14}$$

Ésta es la generalización del caso 2 del Teorema Maestro. (¿Cuál caso especial de la ecuación (3.14) da el caso 2 del Teorema Maestro?)

3.10 Determine el orden asintótico de las soluciones de las ecuaciones de recurrencia siguientes. Puede suponer que $T(1) = 1$, la recurrencia es para $n > 1$ y c es alguna constante positiva. En algunos casos se necesitará la ecuación (3.14), puede usarla sin tener que demostrarla.

a. $T(n) = T(n/2) + c \lg n$.
b. $T(n) = T(n/2) + cn$.
c. $T(n) = 2T(n/2) + cn$.
d. $T(n) = 2T(n/2) + cn \lg n$.
e. $T(n) = 2T(n/2) + cn^2$.

★ **3.11** Considere la ecuación de recurrencia de recorta y serás vencido

$$T(n) = b_1 T(n-1) + b_2 T(n-2) + \cdots + b_k T(n-k) + f(n) \quad \text{para } n \geq k \tag{3.15}$$

para alguna constante $k \geq 2$. Los coeficientes b_i son no negativos; algunos podrían ser cero. Por ejemplo, la recurrencia de Fibonacci, ecuación (1.13), corresponde a $k = 2$, $b_1 = b_2 = 1$ y $f(n) = 0$.

La *ecuación característica* de la ecuación de recurrencia anterior es

$$x^k - b_1 x^{k-1} - b_2 x^{k-2} - \cdots - b_k = 0. \tag{3.16}$$

★ **a.** (Esta parte requiere cálculo avanzado y teoría de polinomios.) Demuestre que la ecuación (3.16) tiene exactamente una raíz real positiva, y que esa raíz es mayor que 1 si y sólo si $(b_1 + \cdots + b_k) > 1$. También, demuestre que la magnitud de cualquier raíz es igual o menor que la de la raíz positiva.

b. Suponga que r es una solución de la ecuación (3.16). Demuestre que $T(n) = r^n$ es una solución de la ecuación (3.16) si $f(n) = 0$ y los casos base son $T(i) = r^i$ para $0 \leq i < k$.

c. Sea r la solución positiva de la ecuación (3.16). Llegue a la conclusión de que si $(b_1 + \cdots + b_k) > 1$ y $T(i) \geq 1$ para $0 \leq i < k$ y $f(n) \geq 0$, entonces $T(n) \in \Omega(r^n)$. Se pueden usar partes anteriores de este ejercicio aunque no se hayan demostrado.

d. Defina $\phi = \frac{1}{2}(1 + \sqrt{5})$; ésta se conoce como la *Razón Dorada* y es aproximadamente 1.618. Demuestre que la solución de la recurrencia de Fibonacci, ecuación (1.13), está en $\Theta(\phi^n)$. Se pueden usar partes anteriores de este ejercicio aunque no se hayan demostrado.

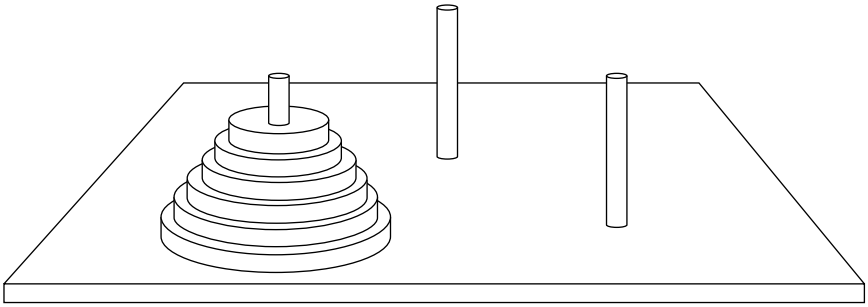


Figura 3.12 Torres de Hanoi

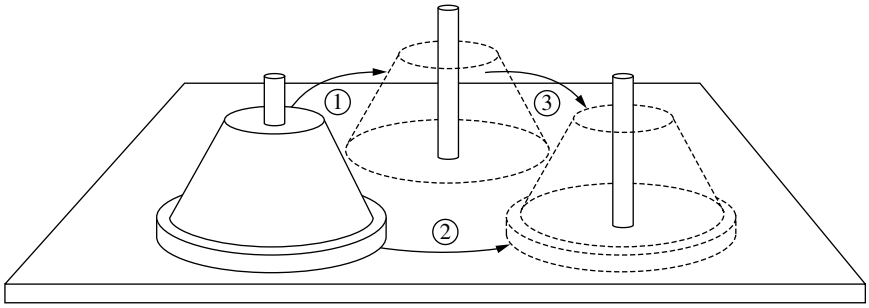


Figura 3.13 Cómo mover los discos

Problemas adicionales

3.12 El problema de las Torres de Hanoi se usa con frecuencia como ejemplo al enseñar recursión. Seis discos de diferente tamaño están ensartados en una varilla y ordenados por tamaño, con el más grande hasta abajo, como se muestra en la figura 3.12. Hay dos varillas vacías. El problema consiste en pasar todos los discos a la tercera varilla moviendo sólo uno a la vez y sin colocar un disco encima de uno más pequeño. Se puede usar la segunda varilla para movimientos intermedios. La solución común mueve recursivamente todos los discos menos el último de la varilla inicial a la varilla intermedia, luego mueve el disco que queda en la varilla inicial a la varilla de destino, luego mueve recursivamente todos los demás discos de la varilla intermedia a la varilla de destino. Los tres pasos se ilustran en la figura 3.13 y se describen en el procedimiento siguiente.

```
hanoi(numeroDeDiscos, inicio, destino, intermedia)
/** Objetivo: mover numeroDeDiscos del tope de la varilla inicial
 * al tope de la varilla de destino, usando la intermedia como búfer. */
   if(numeroDeDiscos > 0)
     hanoi(numeroDeDiscos−1, inicio, intermedia, destino);
     Pasar disco superior de varilla inicio a varilla destino.
     hanoi(numeroDeDiscos−1, intermedia, destino, inicio);
   return
```

Escriba una ecuación de recurrencia para el número de movimientos efectuados, y resuélvala.

3.13 Considere un árbol general *T* (sección 2.3.4) en el que cada vértice v tiene asociado un *peso*, `v.peso`. Un *conjunto independiente* de vértices es un conjunto *I* tal que no existe una arista en *T* entre cualesquier dos vértices que estén en *I*; dicho de otro modo, si el vértice $v \in I$, entonces ni el padre de v ni ninguno de los hijos de v están en *I*. El *peso* de un conjunto de vértices es la sumatoria de sus pesos individuales. El objetivo de este ejercicio es diseñar una función que calcule el peso máximo de cualquier conjunto independiente de vértices del árbol *T*. (Aunque su función no necesita identificar un conjunto independiente que tenga ese peso máximo, será fácil modificarlo para que lo haga.)

La clave para un diseño eficiente es considerar dos colecciones restringidas de conjuntos independientes para *T*: los que *incluyen* la raíz de *T* y los que *excluyen* la raíz de *T*. Denotemos con `tomarPeso` el peso máximo de cualquier conjunto independiente en *T* que *incluya* la raíz de *T*, y denotemos con `dejarPeso` el peso máximo de cualquier conjunto independiente en *T* que *excluya* la raíz de *T*.

a. Dé una definición recursiva de `tomarPeso` para *T* en términos de `raiz(T).peso` y los valores de `tomarPeso` y `dejarPeso` para los subárboles principales de *T*.
b. Dé una definición recursiva de `dejarPeso` para *T* en términos de los valores de `tomarPeso` y `dejarPeso` para los subárboles principales de *T*.
c. Diseñe una función (puede usar pseudocódigo claro, e incluso es preferible) basada en el esqueleto de recorrido de árbol de la figura 2.13 que calcule `tomarPeso` y `dejarPeso`. Utilice una clase organizadora con esos dos campos para que su función pueda devolver ambas cantidades. Si tiene cuidado, no necesitará arreglos ni variables globales.
d. Analice las necesidades de tiempo y espacio de su función.

Notas y referencias

Según Perlis (1978) McCarthy abogó por que el diseño de Algol 60 incluyera recursión. La importancia de la recursión en el diseño de programas se destaca en Roberts (1997), donde hay un tratamiento exhaustivo del tema.

Gries (1981) se ocupó de demostrar la corrección de programas y de idear técnicas para escribir programas con mayores posibilidades de ser correctos. Hantler y King (1976) es una reseña de técnicas tanto formales como informales para demostrar que un programa es correcto. Sethi (1996) describe reglas para demostrar corrección parcial con cierto detalle. Kingston (1997) considera técnicas de demostración para algoritmos. De Millo, Lipton y Perlis (1979) comentan las dificultades técnicas de demostrar corrección. Grassmann y Tremblay (1966) analizan la inducción con muchos conjuntos distintos del de los números naturales.

Hay gran cantidad de artículos acerca de Sisal, un lenguaje de asignación única para programación en paralelo; uno de los primeros fue Oldehoeft, Cann y Allan (1986). Cytron, Ferrante, Rosen, Wegman y Zadeck (1991) exploran las ventajas de la forma de asignación única para el análisis de programas, y describen un algoritmo para convertir un procedimiento a una forma es-

tática de asignación única. Esta forma es ahora una herramienta muy utilizada en la optimización de compiladores y paralelización automática de código.

El uso de árboles de recursión y el Teorema Maestro (teorema 3.17) para evaluar ecuaciones de recurrencia se basa en Cormen, Leiserson y Rivest (1990). Aho, Hopcroft y Ullman (1983) contiene un tratamiento excelente de la estructura de las soluciones de ecuaciones de recurrencia. Si busca herramientas matemáticas más avanzadas para análisis de algoritmos, vea Purdom y Brown (1985), Lueker (1980) y Greene y Knuth (1990).

4

Ordenamiento

4.1 Introducción

En este capítulo estudiaremos varios algoritmos para ordenar, es decir, acomodar en orden los elementos de un conjunto. El problema de ordenar un conjunto de objetos fue uno de los primeros problemas que se estudiaron intensamente en las ciencias de la computación. Muchas de las aplicaciones más conocidas del paradigma de diseño de algoritmos Divide y Vencerás son algoritmos de ordenamiento. Durante los años sesenta, cuando el procesamiento comercial de datos se automatizó en gran escala, el programa de ordenamiento era el que se ejecutaba con mayor frecuencia en muchas instalaciones de cómputo. Una compañía de software se mantuvo operando durante años gracias a que tenía un programa de ordenamiento mejor. Con el hardware actual, los aspectos de desempeño del ordenamiento han cambiado un poco. En los años sesenta, la transferencia de datos entre almacenamiento lento (cinta o disco) y la memoria principal era un importante cuello de botella del desempeño. La memoria principal era del orden de 100,000 bytes y los archivos a procesar eran varios órdenes de magnitud mayores. La atención se concentraba en los algoritmos para efectuar este tipo de ordenamiento. Hoy, las memorias principales 1,000 veces mayores (o sea, de 100 megabytes) son cosa común, y las hay 10,000 veces mayores (de unos cuantos gigabytes), de modo que la mayor parte de los archivos cabe en la memoria principal.

Hay varias razones de peso para estudiar los algoritmos de ordenamiento. La primera es que tienen utilidad práctica porque el ordenamiento es una actividad frecuente. Así como tener las entradas de un directorio telefónico o de un diccionario en orden alfabético facilita su uso, el trabajo con conjuntos grandes de datos en las computadoras se facilita si los datos están ordenados. La segunda es que se ha ideado una buena cantidad de algoritmos para ordenar (más de los que cubriremos aquí), y si el lector estudia varios de ellos se convencerá del hecho de que es posible enfocar un problema dado desde muchos puntos de vista distintos. El tratamiento de los algoritmos en este capítulo deberá dar al lector algunas ideas acerca de cómo se puede mejorar un algoritmo dado y cómo escoger entre varios algoritmos. La tercera es que el ordenamiento es uno de los pocos problemas para los que es fácil deducir cotas inferiores firmes del comportamiento en el peor caso y en el caso promedio. Las cotas son firmes en el sentido de que existen algoritmos que efectúan aproximadamente la cantidad mínima de trabajo especificada. Por ello, tenemos algoritmos de ordenamiento prácticamente óptimos.

Al describir la mayor parte de los algoritmos, supondremos que el conjunto a ordenar está almacenado en forma de arreglo, de modo que se pueda acceder en cualquier momento a un elemento en cualquier posición; esto se denomina *acceso aleatorio*. No obstante, algunos de los algoritmos también son útiles para ordenar archivos y listas ligadas. Si el acceso al conjunto es exclusivamente secuencial, empleamos el término *sucesión* para hacer hincapié en que la estructura podría ser una lista ligada o un archivo secuencial, no sólo un arreglo. Si definimos un arreglo dentro del intervalo de índices $0, \ldots, n-1$, un *intervalo* o *subintervalo* de ese arreglo será una sucesión continua de elementos que está entre dos índices dados, `primero` y `ultimo`, tales que $0 \leq$ `primero` y `ultimo` $\leq n-1$. Si `ultimo` $<$ `primero`, decimos que el intervalo está *vacío*.

Suponemos que cada elemento del conjunto a ordenar contiene un identificador, llamado *clave*, que es un elemento de algún conjunto linealmente ordenado y que es posible comparar dos claves para determinar cuál es mayor o que son iguales. Siempre ordenamos las claves en orden no decreciente. Cada elemento del conjunto podría contener otra información además de la clave. Si las claves se reacomodan durante el proceso de ordenamiento, la información asociada también se reacomodará de manera acorde, pero a veces sólo hablaremos de las claves sin mencionar explícitamente el resto del elemento.

Todos los algoritmos que consideraremos en las secciones 4.2 a 4.10 pertenecen a la clase de algoritmos de ordenamiento que podrían comparar claves (y copiarlas) pero no deben aplicar otras operaciones a las claves. Decimos que éstos son "algoritmos que ordenan comparando claves", o "algoritmos basados en comparación". La medida primordial del trabajo que se usa para analizar algoritmos de esta clase es el número de comparaciones de claves. En la sección 4.7 se establecen cotas inferiores para el número de comparaciones que efectúa este tipo de algoritmos. En la sección 4.11 se tratan algoritmos de ordenamiento que pueden efectuar operaciones distintas de la comparación de claves, y para los cuales son apropiadas otras medidas del trabajo.

Los algoritmos de este capítulo se denominan *ordenamientos internos* porque se supone que los datos están en la memoria de acceso aleatorio de alta velocidad de la computadora. Surgen diferentes aspectos de desempeño cuando se desea ordenar conjuntos de datos tan grandes que no caben en la memoria. Los algoritmos para ordenar grandes conjuntos de datos almacenados en dispositivos de almacenamiento externos más lentos, con restricciones sobre la forma de acceder a los datos, se denominan *ordenamientos externos*. En las Notas y referencias al final del capítulo se dan fuentes de tales algoritmos.

Al analizar algoritmos de ordenamiento, consideraremos qué tanto espacio adicional emplean (además del que ocupan las entradas). Si la cantidad de espacio extra es constante con respecto al tamaño de las entradas, decimos que el algoritmo opera *en su lugar*.

A fin de que los algoritmos sean lo más claros posible, usaremos `Elemento` y `Clave` como identificadores de tipo, pero trataremos a `Clave` como un tipo numérico en cuanto a que usaremos los operadores relacionales "$=, \neq, <$", etc. Si en el libro aparece una expresión de comparación de claves, como "`E[i].clave < x`", y los tipos reales no son numéricos (**`String`**, por ejemplo), la sintaxis de Java requerirá una invocación de método, como "`menor(E[i].clave, x)`". Esto también es necesario en muchos otros lenguajes.

Detalle de Java: Empleando la interfaz **`Comparable`** de Java, es posible escribir un procedimiento capaz de comparar una amplia variedad de tipos de claves. El nombre de tipo `Clave` se sustituiría por la palabra reservada **`Comparable`**. En el apéndice A se dan algunos detalles. Recuerde que un arreglo con elementos de tipo `Elemento` se declara como

```
Elemento[] nombreArreglo;
```

en Java.

4.2 Ordenamiento por inserción

Ordenamiento por inserción es un buen algoritmo de ordenamiento para comenzar porque la idea en la que se basa es natural y general, sus análisis de peor caso y comportamiento promedio son fáciles de efectuar. También se usa como parte de un algoritmo de ordenamiento más rápido que describiremos en la sección 4.10.

4.2.1 La estrategia

Partimos de una sucesión E de n elementos en orden arbitrario, como ilustra la figura 4.1. (Ordenamiento por inserción se puede usar con claves de cualquier conjunto ordenado linealmente, pero en el caso de las ilustraciones de palitos podemos pensar que las claves son las alturas de los palitos, que son los elementos.)

Supóngase que hemos ordenado algún segmento inicial de la sucesión. La figura 4.2 muestra una instantánea de la sucesión una vez que se han ordenado los cinco elementos del extremo

Figura 4.1 Elementos en desorden

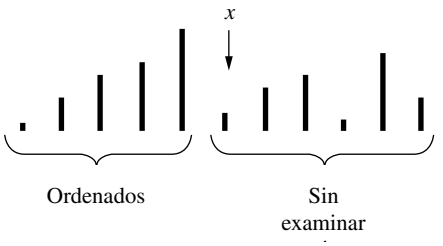


Figura 4.2 Elementos parcialmente ordenados

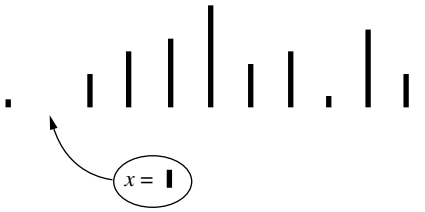


Figura 4.3 Inserción de *x* en el orden correcto

izquierdo. El paso general consiste en incrementar la longitud del segmento ordenado insertando el siguiente elemento en el lugar correcto.

Sea *x* el siguiente elemento a insertar en el segmento ordenado, es decir, *x* es al elemento de la extrema izquierda del segmento que todavía no se examina. Primero "hacemos a un lado" a *x* (es decir, lo copiamos en una variable local), dejando una *vacante* en su antigua posición. Luego comparamos repetidamente *x* con el elemento que está inmediatamente a la izquierda de la vacante y, mientras *x* sea menor, pasaremos ese elemento a la vacante, dejando una vacante en el lugar donde estaba; es decir, la vacante se desplaza una posición hacia la izquierda. Este proceso se detiene cuando se acaban los elementos a la izquierda de la vacante actual, o cuando el elemento que está a la izquierda de la vacante actual es menor o igual que *x*. En ese momento, insertamos *x* en la vacante, como se muestra en la figura 4.3. Para poner en marcha el algoritmo, basta con observar que el primer elemento sólo se puede considerar como un segmento ordenado. Al formalizar esto para tener un procedimiento, suponemos que la sucesión es un arreglo; sin embargo, la idea funciona también con listas y otras estructuras secuenciales.

```
int desplaVac(Elemento[] E, int vacante, Clave x)
```

Condición previa: `vacante` no es negativa.

Condiciones posteriores: Sea `posX` el valor devuelto al invocador. Entonces:

1. Los elementos de E cuyos índices son menores que `posX` están en sus posiciones originales y sus claves son menores o iguales que x.
2. Los elementos de E que están en las posiciones `posX` + 1,..., `vacante` son mayores que x y se desplazaron una posición a la izquierda respecto a la posición que ocupaban cuando se invocó `desplaVac`.

Figura 4.4 Especificaciones de `desplaVac`

4.2.2 El algoritmo y su análisis

Ahora presentaremos los pormenores del procedimiento para ordenar. La tarea de la subrutina `desplaVac`(E, `vacante`, x) es desplazar elementos hasta que la vacante esté en la posición correcta para colocar x entre los elementos ordenados. El procedimiento devuelve el índice de la vacante, digamos `posX`, al invocador. Las condiciones previas y posteriores se plantean en la figura 4.4. En otras palabras, `desplaVac` efectúa la transición de la figura 4.2 a la 4.3. Ahora `ordenInsercion` sólo tiene que invocar repetidamente a `desplaVac`, formando un segmento ordenado cada vez más largo en el extremo izquierdo, hasta que todos los elementos estén en ese segmento.

El procedimiento `desplaVac` adopta la forma representativa de las *rutinas de búsqueda generalizada* (definición 1.12). Si no hay más datos que examinar, fracasar; si hay más datos, examinar uno, y si es el que estamos buscando, tener éxito; en caso contrario, continuar con los datos no examinados. Puesto que hay dos casos para terminar, no sería conveniente usar un ciclo **`while`**, a menos que se usara un **`break`** con uno o más de los casos para terminar. La formulación recursiva es sencilla.

```
int desplaVacRec(Elemento[] E, int vacante, Clave x)
    int posX;
1.  if (vacante == 0)
2.     posX = vacante;
3.  else if (E[vacante−1].clave ≤ x)
4.     posX = vacante;
5.  else
6.     E[vacante] = E[vacante−1];
7.     posX = desplaVacRec(E, vacante−1, x);
8.  return posX;
```

Para verificar que estamos usando recursión correctamente en la línea 7, observamos que la invocación recursiva está operando con un intervalo más pequeño, y que su segundo argumento no es negativo, con lo que se satisface la condición previa (planteada en la figura 4.4). (Recomendamos al lector verificar la cadena de razonamiento que nos dice que `vacante` − 1 no es negativa; ¿por qué no puede ser negativa?) Ahora es sencillo demostrar la corrección si recordamos que podemos *suponer* que la invocación recursiva de la línea 7 logra su objetivo.

Aunque el procedimiento para `desplaVacRec` es muy sencillo, si visualizamos el rastreo de activación para el *n*-ésimo elemento de *E* a insertar, nos daremos cuenta de que la profundidad de la recursión, o la pila de marcos, podría crecer hasta un tamaño *n*. Esto podría ser indeseable si *n* es grande. Por tanto, éste es un caso en que conviene convertir la recursión en una iteración, una vez que hayamos constatado que todo funciona correctamente. (Tratar de optimar un programa que no funciona ciertamente sería fútil.) El objetivo no es tanto ahorrar tiempo como ahorrar espacio. En realidad muchos compiladores, si se les pide optimar a `desplaVacRec`, efectuarán la transformación automáticamente. El algoritmo completo que sigue incluye la versión de `desplaVac` codificada iterativamente.

Algoritmo 4.1 Ordenamiento por inserción

Entradas: *E*, un arreglo de elementos, y $n \geq 0$, el número de elementos. El intervalo de los índices es $0, \ldots, n - 1$.

Salidas: *E*, con los elementos en orden no decreciente según sus claves.

Comentario: Las especificaciones de la subrutina `desplaVac` se dan en la figura 4.4.

```
void ordenInsercion(Elemento[] E, int n)
     int indicex;
     for (indicex = 1; indicex < n; indicex ++)
         Elemento actual = E[indicex];
         Clave x = actual.clave;
         int posX = desplaVac(E, indicex, x);
         E[posX] = actual;
     return;

int desplaVac(Elemento[] E, int indicex, Clave x)
     int vacante, posX;
     vacante = indicex;
     posX = 0; // Suponemos fracaso.
     while (vacante > 0)
         if (E[vacante−1].clave ≤ x)
             posX = vacante; // Éxito.
             break;
         E[vacante] = E[vacante−1];
         vacante --; // Seguir buscando.
     return posX;
```

Complejidad de peor caso

Para el análisis, usamos *i* en lugar de `indicex`. Para cada valor de *i*, el número máximo de comparaciones que pueden efectuarse (en una invocación de la rutina iterativa `desplaVac` o en una invocación de nivel más alto de la rutina recursiva `desplaVacRec`) es *i*. Por tanto, el total es

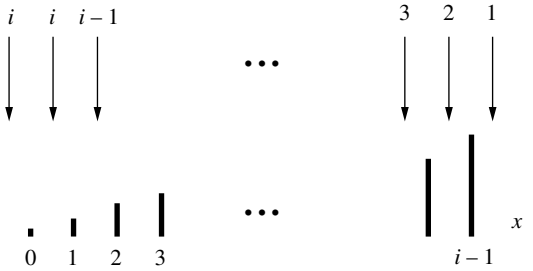


Figura 4.5 Número de comparaciones necesarias para determinar la posición de x

$$W(n) \leq \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Observe que hemos establecido una cota superior para el comportamiento de peor caso; hay que pensar un momento para verificar que en verdad existen entradas con las que se efectúan $n(n-1)/2$ comparaciones. Uno de esos peores casos es cuando las claves están en orden inverso (es decir, decreciente). Así,

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2).$$

Comportamiento promedio

Suponemos que todas las permutaciones de las claves son entradas igualmente verosímiles. Primero determinaremos cuántas comparaciones de claves se efectúan en promedio para insertar un elemento nuevo en el segmento ordenado, es decir, en una invocación de `desplaVac` con cualquier valor específico de i (que usamos en vez de `indicex`). Para simplificar el análisis, suponemos que todas las claves son distintas. (El análisis es muy similar al que se efectúa con el algoritmo Búsqueda Secuencial en el capítulo 1.)

Hay $i + 1$ posiciones en las que puede colocarse x. La figura 4.5 muestra cuántas comparaciones se efectúan dependiendo de la posición.

La probabilidad de que x vaya en cualquier posición específica es $1/(i + 1)$. (Esto depende del hecho de que el algoritmo todavía no ha examinado a x. Si el algoritmo hubiera tomado antes alguna decisión con base en el valor de x, no podríamos suponer por fuerza que x es uniformemente aleatorio con respecto a las i primeras claves.) Así pues, el número medio de comparaciones que se efectúan en `desplaVac` para encontrar la posición del i-ésimo elemento es

$$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{1}{i+1}(i) = \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}.$$

Ahora obtenemos la sumatoria para las $n - 1$ inserciones:

$$A(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n}\frac{1}{j},$$

donde sustituimos $j = i + 1$ para obtener la última sumatoria. Ya vimos, por la ecuación (1.16), que $\sum_{j=1}^{n} (1/j) \approx \ln n$, podemos incorporar el 1 que está antes de la sumatoria para hacer que el límite inferior $j = 1$. Olvidándonos de los términos de orden inferior, tenemos

$$A(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

Espacio

Es evidente que Ordenamiento por inserción es un ordenamiento en su lugar si usamos la versión iterativa de `desplaVac`. Con la versión recursiva, la pila de marcos puede crecer hasta $\Theta(n)$.

4.2.3 Cotas inferiores para el comportamiento de ciertos algoritmos de ordenamiento

Pensemos que el elemento cuya clave es x el cual ocupa la posición "vacante" del arreglo, mientras Ordenamiento por inserción compara x con la clave que está a su izquierda. Entonces, después de cada comparación, Ordenamiento por inserción no cambiará de lugar ningún elemento o simplemente intercambiará dos elementos adyacentes. Demostraremos que todos los algoritmos de ordenamiento que efectúan semejante traslado "local" limitado de elementos, después de cada comparación, deben efectuar aproximadamente la misma cantidad de trabajo que Ordenamiento por inserción.

Una permutación de n elementos se puede describir con una función uno a uno del conjunto $N = \{1, 2, \ldots, n\}$ sobre sí mismo. Existen $n!$ permutaciones distintas de n elementos. Sean x_1, $x_2, \ldots, x_n$ los elementos de la sucesión no ordenada E. A fin de simplificar la notación en este análisis, supóngase que los elementos a ordenar están en las posiciones $1, \ldots, n$ de E, no en $0, \ldots, n - 1$. Existe una permutación π tal que, para $1 \leq i \leq n$, $\pi(i)$ es la posición correcta de x_i una vez que la sucesión está ordenada. Sin perder generalidad, podemos suponer que las claves son los enteros $1, 2, \ldots, n$, ya que podemos usar 1 en lugar de la clave más pequeña, 2 en lugar de la clave más pequeña de las restantes y así sucesivamente, sin tener que modificar las instrucciones ejecutadas por el algoritmo. Entonces, la entrada sin ordenar es $\pi(1), \pi(2), \ldots, \pi(n)$. Por ejemplo, consideremos la sucesión de entrada 2, 4, 1, 5, 3. $\pi(1) = 2$ implica que la primera clave, 2, debe ir en la segunda posición, lo cual es obvio. $\pi(2) = 4$ porque la segunda clave, 4, debe ir en la cuarta posición, y así sucesivamente. Identificaremos la permutación π con la sucesión $\pi(1), \pi(2), \ldots, \pi(n)$.

Una *inversión* de la permutación π es un par $(\pi(i), \pi(j))$ tal que $i < j$ y $\pi(i) > \pi(j)$. Si $(\pi(i), \pi(j))$ es una inversión, las claves i-ésima y j-ésima de la sucesión están en desorden una respecto a la otra. Por ejemplo, la permutación 2, 4, 1, 5, 3 tiene cuatro inversiones: (2, 1), (4, 1), (4, 3) y (5, 3). Si un algoritmo de ordenamiento elimina cuando más una inversión después de cada comparación de claves (por ejemplo, al intercambiar elementos adyacentes, como hace Ordenamiento por inserción), entonces el número de comparaciones efectuadas con la entrada $\pi(1), \pi(2), \ldots, \pi(n)$ será por lo menos el número de inversiones de π. Por ello, investigaremos las inversiones.

Es fácil demostrar que existe una permutación con $n(n - 1)/2$ inversiones. (¿Cuál permutación?) Por tanto, el comportamiento de peor caso de cualquier algoritmo de ordenamiento que elimina cuando más una inversión en cada comparación de claves deberá estar en $\Omega(n^2)$.

Para obtener una cota inferior del número medio de comparaciones efectuadas por tales algoritmos de ordenamiento, calculamos el número medio de inversiones que hay en las permutaciones. Cada permutación π se puede aparear con su *permutación transpuesta* $\pi(n), \pi(n - 1), \ldots,$

$\pi(1)$. Por ejemplo, la transpuesta de 2, 4, 1, 5, 3 es 3, 5, 1, 4, 2. Cada permutación tiene una transpuesta única y es distinta de su transpuesta (para $n > 1$). Sean i y j enteros entre 1 y n, y supóngase que $j < i$. Entonces (i, j) es una inversión en una y sólo una de las permutaciones π y transpuesta de π. Existen $n(n-1)/2$ pares de enteros semejantes. Por tanto, cada par de permutaciones tiene $n(n-1)/2$ permutaciones en conjunto, y por ende un promedio de $n(n-1)/4$. Así, en total, el número medio de inversiones que hay en una permutación es de $n(n-1)/4$, hemos demostrado el teorema siguiente.

Teorema 4.1 Cualquier algoritmo que ordena por comparación de claves y elimina cuando más una inversión después de cada comparación deberá efectuar al menos $n(n-1)/2$ comparaciones en el peor caso y al menos $n(n-1)/4$ comparaciones en promedio (con n elementos). □

Puesto que Ordenamiento por inserción efectúa $n(n-1)/2$ comparaciones de claves en el peor caso y aproximadamente $n^2/4$ en promedio, es prácticamente lo mejor que podemos lograr con cualquier algoritmo que opere "localmente", digamos intercambiando sólo elementos adyacentes. Desde luego, a estas alturas no es obvio que alguna otra estrategia pueda funcionar mejor, pero si existen algoritmos significativamente más rápidos deberán trasladar elementos más de una posición a la vez.

4.3 Divide y vencerás

El principio en que se basa el paradigma de diseño de algoritmos Divide y vencerás es que (a menudo) es más fácil resolver varios casos pequeños de un problema que uno grande. Los algoritmos de las secciones 4.4 a 4.8 emplean el enfoque de Divide y vencerás: *dividen* el problema en ejemplares más pequeños del mismo problema (en este caso, conjuntos más pequeños a ordenar), luego resuelven (*vencen*) los ejemplares más pequeños de forma recursiva (o sea, empleando el mismo método) y por último *combinan* las soluciones para obtener la solución correspondiente a la entrada original. Para escapar de la recursión, resolvemos directamente algunos casos pequeños del problema. En contraste, Ordenamiento por inserción se limitó a "recortar" un elemento para crear un subproblema.

Ya vimos un ejemplo excelente de Divide y vencerás: Búsqueda binaria (sección 1.6). El problema principal se dividió en dos subproblemas, uno de los cuales ni siquiera se tenía que resolver.

En general, podemos describir Divide y vencerás con el esqueleto de procedimiento de la figura 4.6.

Para diseñar un algoritmo de Divide y vencerás específico, debemos especificar las subrutinas `resolverDirectamente`, `dividir` y `combinar`. El número de casos más pequeños en los que se divide la entrada es k. Con una entrada de tamaño n, sea $B(n)$ el número de pasos efectuados por `resolverDirectamente`, sea $D(n)$ el número de pasos efectuados por `dividir`, y sea $C(n)$ el número de pasos efectuados por `combinar`. Entonces, la forma general de la ecuación de recurrencia que describe la cantidad de trabajo efectuada por el algoritmo es

$$T(n) = D(n) + \sum_{i=1}^{k} T(\textit{tamaño}(I_i)) + C(n) \qquad \text{para } n > \texttt{pequeño}$$

```
resolver(I)
    n = tamaño(I);
    if (n ≤ pequeño)
        solucion = resolverDirectamente(I);
    else
        dividir I en I_1,..., I_k.
        para cada i ∈ {I_1,..., I_k}:
            S_i = resolver(I_i);
        solucion = combinar(S_1,..., S_k);
    return solucion;
```

Figura 4.6 El esqueleto de Divide y vencerás

con los casos base $T(n) = B(n)$ para $n \leq$ `pequeño`. En muchos algoritmos Divide y vencerás, el paso de dividir o bien el paso de combinar es muy sencillo, y la ecuación de recurrencia para T es más simple que la forma general. El Teorema maestro (teorema 3.17) da soluciones para una amplia gama de ecuaciones de recurrencia de Divide y vencerás.

Quicksort y Mergesort, los algoritmos de ordenamiento que presentamos en las próximas secciones, difieren en la forma en que dividen el problema y luego combinan las soluciones, o subconjuntos ordenados. Quicksort se caracteriza como "división difícil, combinación fácil", mientras que Mergesort se caracteriza como "división fácil, combinación difícil". Fuera del procesamiento que requieren las invocaciones de procedimientos, veremos que el "trabajo real" se efectúa en la sección "difícil". Ambos procedimientos de ordenamiento tienen subrutinas para realizar su sección "difícil", y tales subrutinas son útiles por derecho propio. En el caso de Quicksort, el "caballito de batalla" es `partir`, y es el paso `dividir` del esqueleto general; el paso `combinar` no hace nada. En el caso de Mergesort, el "caballito de batalla" es `fusionar`, y es el paso `combinar`; el paso `dividir` sólo efectúa un cálculo sencillo. Ambos algoritmos dividen el problema en dos subproblemas. Sin embargo, en Mergesort los problemas son de tamaño comparable (más o menos un elemento), mientras que en Quicksort no se garantiza una división pareja. Esta diferencia da pie a características de desempeño muy distintas, que descubriremos durante el análisis de los respectivos algoritmos.

En el nivel más alto, HeapSort (sección 4.8) no es un algoritmo Divide y vencerás, pero usa operaciones de montón que pertenecen a la categoría Divide y vencerás. La forma acelerada de Heapsort emplea un algoritmo Divide y vencerás más avanzado.

En capítulos posteriores encontraremos la estrategia Divide y vencerás en numerosos problemas. En el capítulo 5 la aplicaremos al problema de hallar el elemento que es la mediana de un conjunto. (El problema general se denomina problema de selección.) En el capítulo 6 usaremos Divide y vencerás en la forma de árboles de búsqueda binaria y sus versiones equilibradas, los árboles rojinegros. En el capítulo 9 aplicaremos la estrategia a problemas de caminos en grafos, como el cierre transitivo. En el capítulo 12 la usaremos en varios problemas de matrices y vectores. En el capítulo 13 la aplicaremos al coloreado aproximado de grafos. En el capítulo 14 reaparecerá en una forma un poco distinta para la computación en paralelo.

4.4 Quicksort

Quicksort es uno de los primeros algoritmos Divide y vencerás que se descubrieron; fue publicado por C.A.R. Hoare en 1962 y sigue siendo uno de los más rápidos en la práctica.

4.4.1 La estrategia Quicksort

La estrategia de Quicksort consiste en reacomodar los elementos a ordenar de modo que todas las claves "pequeñas" precedan a las claves "grandes" en el arreglo (la parte de "división difícil"). Luego Quicksort ordena los dos subintervalos de claves "pequeñas" y "grandes" recursivamente, el resultado es que todo el arreglo queda ordenado. Si la implementación se hace con arreglos, no hay nada que hacer en el paso de "combinación", pero Quicksort también puede funcionar con listas (véase el ejercicio 4.22), en cuyo caso el paso de "combinación" concatena las dos listas. Por sencillez, describiremos sólo la implementación con arreglos.

Sea E el arreglo de elementos y sean `primero` y `ultimo` los índices de los elementos primero y último, respectivamente, del subintervalo que Quicksort está ordenando actualmente. En el nivel más alto, `primero` $= 0$ y `ultimo` $= n - 1$, donde n es el número de elementos.

El algoritmo Quicksort escoge un elemento, llamado *elemento pivote* y cuya clave es `pivote`, del subintervalo que debe ordenar, y "lo hace a un lado"; es decir, lo coloca en una variable local, dejando una *vacante* en el arreglo. Por el momento, supondremos que se escoge como elemento pivote el elemento de la extrema izquierda del subintervalo.

Quicksort pasa el pivote (sólo el campo `clave`) a la subrutina Partir, que reacomoda *los demás* elementos, encontrando un índice `puntoPartir` tal que:

1. para `primero` $\leq$ `i` $<$ `puntoPartir`, `E[i].clave` $<$ `pivote`;
2. y para `puntoPartir` $<$ `i` $\leq$ `ultimo`, `E[i].clave` $\geq$ `pivote`.

Observe que ahora hay una *vacante* en `puntoPartir`.

Ahora Quicksort deposita el elemento pivote en `E[puntoPartir]`, que es su posición correcta, y hace caso omiso de él en el ordenamiento subsiguiente. (Véase la figura 4.7.) Esto completa el proceso de "dividir", y lo siguiente que hace Quicksort es invocarse a sí mismo recursivamente para resolver los dos problemas creados por Partir.

El procedimiento Quicksort podría optar por partir el arreglo con base en cualquier clave del arreglo entre `E[primero]` y `E[ultimo]`, como paso previo. Sea cual sea el elemento escogido, se coloca en una variable local llamada `pivote` y, si *no* es `E[primero]`, `E[primero]` se coloca en su posición, lo que garantiza que habrá una vacante en `E[primero]` cuando se invoque Partir. En la sección 4.4.4 se exploran otras estrategias para escoger un pivote.

Algoritmo 4.2 Quicksort

Entradas: Arreglo E e índices `primero` y `ultimo`, tales que están definidos elementos `E[i]` para `primero` $i \leq$ `ultimo`.

Salida: `E[primero]`,..., `E[ultimo]` es un reacomodo ordenado de los mismos elementos.

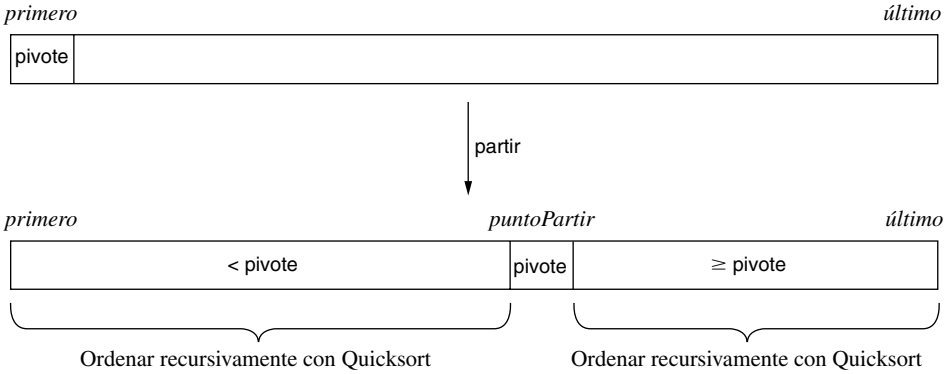


Figura 4.7 Quicksort

```
void quickSort(Elemento[] E, int primero, int ultimo)
    if (primero < ultimo)
        Elemento elementoPivote = E[primero];
        Clave pivote = elementoPivote.clave;
        int puntoPartir = partir(E, pivote, primero, ultimo);
        E[puntoPartir] = elementoPivote;
        quickSort(E, primero, puntoPartir − 1);
        quickSort(E, puntoPartir + 1, ultimo);
    return;
```

4.4.2 La subrutina Partir

Todo el trabajo de comparar claves y cambiar elementos de lugar lo efectúa la subrutina Partir. Ésta puede usar una de varias estrategias que producen algoritmos con diferentes ventajas y desventajas. Aquí presentaremos una y consideraremos otra en los ejercicios. La estrategia gira en la forma de efectuar el reacomodo de los elementos. Una solución muy sencilla consiste en transferir elementos a un arreglo temporal, pero el reto es reacomodarlos en su lugar.

El método para partir que describimos a continuación es en esencia el que Hoare describió originalmente. Como justificación, recordemos que el argumento de cota inferior de la sección 4.2.3 demostró que, si queremos un desempeño mejor que el de Ordenamiento por inserción, es preciso poder trasladar un elemento a muchas posiciones de distancia después de una comparación. Aquí la vacante está inicialmente en `E[primero]`. Puesto que queremos a los elementos pequeños en el extremo izquierdo del intervalo, y queremos trasladar elementos distancias largas siempre que sea posible, es muy lógico comenzar a buscar un elemento pequeño (es decir, un elemento menor que `pivote`) partiendo de `E[ultimo]` hacia atrás. Si lo hallamos, lo trasladamos a la vacante (que estaba en `primero`). Ello deja una nueva vacante en el lugar donde estaba el elemento pequeño, a la cual llamaremos `vacAlta`. La situación se ilustra en los dos primeros diagramas de arreglo de la figura 4.8.

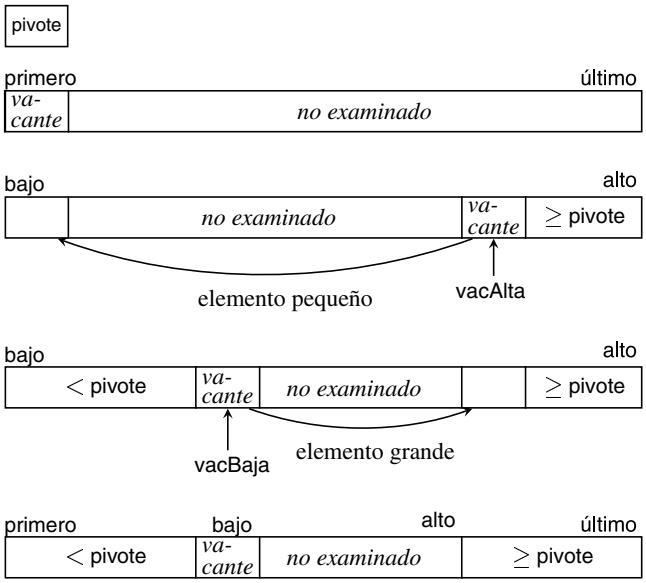


Figura 4.8 Avance de Partir por su primer ciclo

Sabemos que todos los elementos cuyo índice es mayor que `vacAlta` (y hasta `ultimo`) son mayores o iguales que `pivote`. Si es posible, se deberá colocar algún otro elemento grande en `vacAlta`. Una vez más, queremos trasladar los elementos distancias grandes, por lo que en esta ocasión es lógico buscar un elemento *grande* partiendo de `primero + 1` hacia adelante. Una vez que lo hallemos, trasladaremos ese elemento a la vacante (que estaba en `vacAlta`), ello dejará una nueva vacante a la que llamaremos `vacBaja`. Sabemos que todos los elementos cuyo índice es menor que `vacBaja` (y hasta `primero`) son menores que `pivote`.

Por último, actualizamos las variables `bajo` y `alto` como se indica en la última fila de la figura 4.8, en preparación para otro ciclo. Al igual que al principio del primer ciclo, todavía no se han examinado los elementos del intervalo `bajo+1`,..., `alto`, y `E[bajo]` está vacante. Podemos repetir el ciclo que acabamos de describir, buscando un elemento pequeño desde `alto` hacia atrás, transfiriéndolo a la vacante baja, luego buscando un elemento alto desde `bajo+1` hacia adelante y transifiréndolo a `vacAlta`, con lo que se crea una vacante en `vacBaja`, la posición desde la cual se transfirió el elemento grande. En algún momento, `vacBaja` se topará con `vacAlta`, y ello implicará que ya se han comparado todos los elementos con el pivote.

El procedimiento Partir se implementa como una repetición del ciclo que acabamos de describir, empleando subrutinas para organizar el código. La subrutina `extenderRegionGrande` busca hacia atrás a partir del extremo derecho, pasando por alto los elementos grandes hasta encontrar un elemento pequeño y trasladarlo a la vacante del extremo izquierdo o bien hasta llegar a esa vacante sin haber encontrado elementos pequeños. En el segundo caso, se habrá terminado de partir el arreglo. En el primer caso, se devolverá la nueva posición vacante y se invocará la siguiente subrutina. La subrutina `extenderRegionChica` es similar, excepto que busca hacia ade-

lante a partir del extremo izquierdo, pasando por alto los elementos pequeños hasta encontrar un elemento grande el cual traslada a la vacante del extremo derecho, o hasta que se queda sin datos.

En un principio, la región de claves pequeñas (a la izquierda de `bajo`) y la región de claves grandes (a la derecha de `alto`) están vacías, la vacante está en el extremo izquierdo de la región media (que es todo el intervalo en este momento). Cada invocación de una subrutina, sea `extenderRegionGrande` o `extenderRegionChica`, encoge la región media en por lo menos uno, y desplaza la vacante al otro extremo de la región media. Además, las subrutinas garantizan que sólo se colocarán elementos pequeños en la región de claves pequeñas y sólo se colocarán elementos grandes en la región de claves grandes. Esto es evidente por sus condiciones posteriores. Cuando la región media se encoja hasta ocupar sólo una posición, esa posición será la vacante, y se devolverá como `puntoPartir`. Se deja como ejercicio determinar, línea por línea del ciclo **`while`** de `partir`, cuáles son las fronteras de la región media y en qué extremo está la vacante. Aunque el procedimiento de Parir puede "arreglárselas" con menos variables, cada una de las variables que definimos tiene su propio significado, y simplifica la respuesta del ejercicio.

Algoritmo 4.3 Partir

Entradas: Arreglo *E*, `pivote` (la clave en torno a la cual se parte) y los índices `primero` y `ultimo`, tales que están definidos elementos `E[i]` para `primero` $+ 1 \leq i \leq$ `ultimo` y `E[primero]` está vacante. Se supone que `primero` $<$ `ultimo`.

Salidas: Sea `puntoPartir` el valor devuelto. Los elementos que originalmente estaban en `primero+1,...,ultimo` se reacomodan en dos subintervalos, tales que

1. las claves de `E[primero],...,E[puntoPartir` $-$ `1]` son menores que `pivote`, y
2. las claves de `E[puntoPartir+1],...,E[ultimo]` son mayores o iguales que `pivote`.

Además, `primero` $\leq$ `puntoPartir` $\leq$ `ultimo`, y `E[puntoPartir]` está vacante.
Procedimiento: Véase la figura 4.9. ■

Para evitar comparaciones adicionales dentro del ciclo **`while`** de `partir`, no se prueba si `vacAlta` $=$ `vacBaja` antes de la línea 5, lo cual indicaría que ya se colocaron todos los elementos en la partición correcta. Por esto, `alto` podría ser uno menos que `bajo` cuando el ciclo termina, aunque lógicamente deberían ser iguales. Sin embargo, ya no se vuelve a usar `alto` después de que el ciclo termina, así que esta diferencia no causa problemas.

En la figura 4.10 se muestra un ejemplo pequeño. Sólo se muestra el funcionamiento detallado de Partir la primera vez que se invoca. Observe que los elementos más pequeños se acumulan a la izquierda de `bajo` y los elementos mayores se acumulan a la derecha de `alto`.

4.4.3 Análisis de Quicksort

Peor caso

Partir compara cada clave con `pivote`, de modo que si hay k posiciones en el intervalo del arreglo con el que está trabajando, efectuará $k - 1$ comparaciones de claves. (La primera posición está vacante.) Si `E[primero]` tiene la clave más pequeña del intervalo que se está partiendo, entonces `puntoPartir` $=$ `primero`, lo único que se habrá logrado es dividir el intervalo en un subintervalo vacío (claves más pequeñas que `pivote`) y un subintervalo con $k - 1$ elementos. Así pues,

```
int partir(Elemento[] E, Clave pivote, int primero, int ultimo)
   int bajo, alto;
1. bajo = primero; alto = ultimo;
2. while (bajo < alto)
3.    int vacAlta = extenderRegionGrande(E, pivote, bajo, alto);
4.    int vacBaja = extenderRegionChica(E, pivote, bajo+1, vacAlta);
5.    bajo = vacAlta; alto = vacAlta − 1;
6. return bajo; // Éste es puntoPartir.

/** Condición posterior de extenderRegionGrande:
  * El elemento de la extrema derecha de E[vacBaja+1],..., E[alto]
  * cuya clave es < pivote, se transfiere a E[vacBaja] y
  * se devuelve el índice de la posición en la que estaba.
  * Si no hay tal elemento, se devuelve vacBaja.
  */
int extenderRegionGrande(Elemento[] E, Clave pivote, int vacBaja, int alto)
   int vacAlta, actual;
   vacAlta = vacBaja; // Suponer fracaso, clave < pivote.
   actual = alto;
   while (actual > vacBaja)
      if (E[actual].clave < pivote)
           E[vacBaja] = E[actual]; // Éxito.
           vacAlta = actual;
           break;
      actual --; // Seguir buscando.
   return vacAlta;

/** Condición posterior de extenderRegionPequeña: (Ejercicio) */
int extenderRegionChica(Elemento[] E, Clave pivote, int bajo, int vacAlta)
   int vacBaja, actual;
   vacBaja = vacAlta; // Suponer fracaso, clave pivote.
   actual = bajo;
   while (actual < vacAlta)
      if (E[actual].clave ≥ pivote)
           E[vacAlta] = E[actual]; // Éxito.
           vacBaja = actual;
           break;
      actual ++; // Seguir buscando.
   return vacBaja;
```

Figura 4.9 Procedimiento del algoritmo 4.3

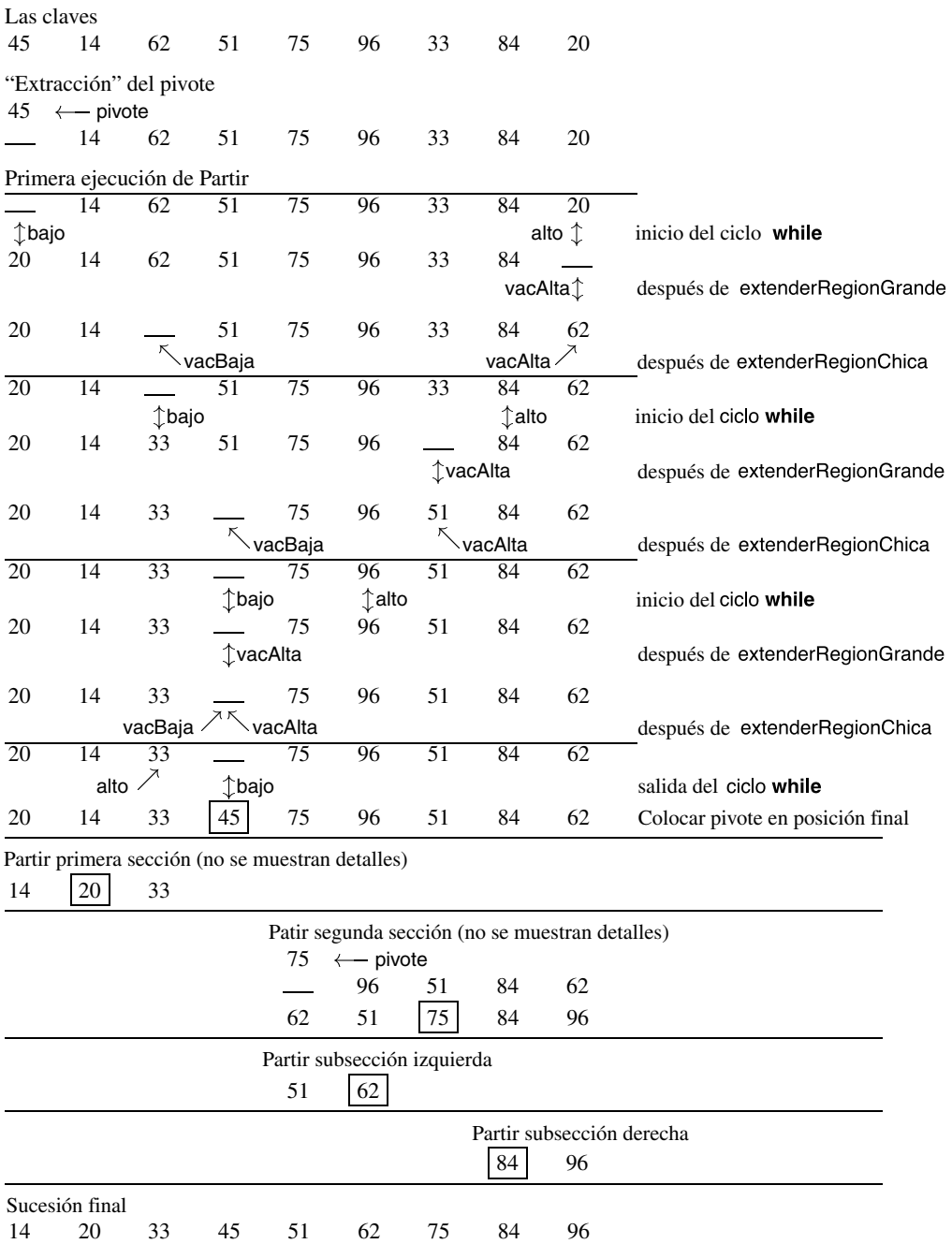


Figura 4.10 Ejemplo de Quicksort

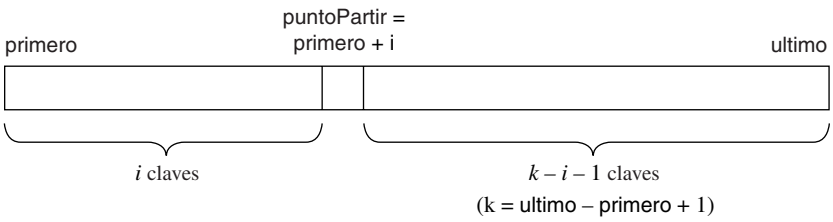


Figura 4.11 Comportamiento promedio de Quicksort

si `pivote` tiene la clave más pequeña en cada ocasión en que se invoca Partir, el número total de comparaciones de clave efectuadas será

$$\sum_{k=2}^{n} (k-1) = \frac{n(n-1)}{2}.$$

Esto es tan malo como Ordenamiento por inserción y Maxsort (ejercicio 4.1). Y, curiosamente, ¡el peor caso se presenta cuando las claves ya están en orden ascendente! ¿Es el nombre Quicksort un caso de publicidad engañosa?

Comportamiento promedio

En la sección 4.2.3 demostramos que si un algoritmo de ordenamiento elimina cuando más una inversión de la permutación de las claves después de cada comparación, deberá efectuar al menos $(n^2 - n)/4$ comparaciones en promedio (teorema 4.1). Sin embargo, Quicksort no tiene esta restricción. El algoritmo Partir puede trasladar un elemento pasando por alto una sección grande del arreglo, eliminando hasta $n - 1$ inversiones con un solo movimiento. Quicksort justifica su nombre por su comportamiento promedio.

Suponemos que las claves son distintas y que todas las permutaciones de las claves son igualmente verosímiles. Sea k el número de elementos contenidos en el intervalo del arreglo que se está ordenando, sea $A(k)$ el número medio de comparaciones de clave que se efectúan con intervalos de este tamaño. Supóngase que la próxima vez que se ejecuta Partir, `pivote` se coloca en la i-ésima posición de este intervalo (figura 4.11), contando desde 0. Partir efectúa $k - 1$ comparaciones de claves, los subintervalos que se ordenarán a continuación tienen i elementos y $k - 1 - i$ elementos, respectivamente.

Es importante para nuestro análisis que, una vez que Partir termina, no se han comparado entre sí ninguna de las dos claves dentro del subintervalo (`primero`,..., `puntoPartir` $-$ 1), de modo que todas las permutaciones de claves dentro de este subintervalo siguen siendo igualmente verosímiles. Lo mismo es cierto para el subintervalo (`puntoPartir` + 1,..., `ultimo`). Esto justifica la recurrencia siguiente.

Todas las posibles posiciones para el punto de partición i son igualmente verosímiles (tienen probabilidad $1/k$), así que, si hacemos $k = n$, tenemos la ecuación de recurrencia

$$A(n) = n - 1 + \sum_{i=0}^{n-1} \frac{1}{n} (A(i) + A(n-1-i)) \qquad \text{para } n \geq 2$$

$$A(1) = A(0) = 0.$$

Una inspección de los términos de la sumatoria nos permite simplificar la ecuación de recurrencia. Los términos de la forma $A(n - 1 - i)$ van desde $A(n - 1)$ hasta $A(0)$, por lo que su sumatoria es igual a la sumatoria de los términos $A(i)$. Entonces podemos desechar los términos en $A(0)$, lo que da

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \qquad \text{para } n \geq 1. \tag{4.1}$$

Esta ecuación de recurrencia es más complicada que las que vimos antes, porque el valor de $A(n)$ depende de todos los valores anteriores. Podemos tratar de aplicar nuestro ingenio para resolver la recurrencia, o bien tratar de adivinar la solución y demostrarla por inducción. Para los algoritmos recurrentes es muy apropiada la segunda técnica, pero se aprende mucho al estudiar ambas, por lo que haremos eso.

Para hacer una conjetura de $A(n)$, consideremos un caso en el que Quicksort funciona muy bien. Supóngase que, cada vez que se ejecuta Partir, divide el intervalo en dos subintervalos iguales. Puesto que sólo estamos haciendo una estimación que nos ayude a conjeturar cuál es la rapidez de Quicksort en promedio, supondremos que el tamaño de los dos intervalos es $n/2$ y no nos preocuparemos por que esta cifra sea entera o no. El número de comparaciones efectuadas se describe con la ecuación de recurrencia

$$Q(n) \approx n + 2Q(n/2).$$

Podemos aplicar el Teorema maestro (teorema 3.17): $b = 2$, $c = 2$, así que $E = 1$ y $f(n) = n^1$. Por tanto, $Q(n) \in \Theta(n \log n)$. Entonces, si `E[primero]` es cercano a la mediana cada vez que se divide el intervalo, el número de comparaciones efectuadas por Quicksort estaría en $\Theta(n \log n)$. Esto es considerablemente mejor que $\Theta(n^2)$. Sin embargo, si todas las permutaciones de las claves son igualmente verosímiles, ¿hay suficientes casos "buenos" como para que afecten el promedio? Demostraremos que sí los hay.

Teorema 4.2 Sea $A(n)$ tal que esté definida por la ecuación de recurrencia (4.1). Entonces, para $n \geq 1$, $A(n) \leq cn \ln n$ para alguna constante c. (*Nota:* Cambiamos a logaritmos naturales para simplificar algunos de los cálculos de la demostración. El valor de c se obtendrá en la demostración.)

Demostración La demostración es por inducción con n, el número de elementos a ordenar. El caso base es $n = 1$. Tenemos $A(1) = 0$ y $c\ 1 \ln 1 = 0$.

Para $n > 1$, suponemos que $A(i) \leq ci \ln(i)$, para $1 \leq i < n$, para la misma constante c planteada en el teorema. Por la ecuación (4.1) y la hipótesis de inducción,

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \leq n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} ci \ln(i).$$

Podemos acotar la sumatoria integrando (véase la ecuación 1.16):

$$\sum_{i=1}^{n-1} ci \ln(i) \leq c \int_1^n x \ln x \, dx.$$

Si usamos la ecuación (1.15) de la sección 1.3.2 obtendremos

$$\int_1^n x \ln x \, dx = \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2$$

así que

$$A(n) \leq n - 1 + \frac{2c}{n}\left(\frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2\right)$$

$$= cn \ln n + n\left(1 - \frac{c}{2}\right) - 1.$$

Para demostrar que $A(n) \leq cn \ln n$, basta con demostrar que los términos segundo y tercero son negativos o cero. El segundo término es cero o menor que cero si $c \geq 2$. Así pues, podemos hacer $c = 2$ y concluir que $A(n) \leq 2\, n \ln n$. □

Un análisis similar muestra que $A(n) > cn \ln n$ para cualquier $c < 2$. Puesto que $\ln n \approx 0.693 \lg n$, tenemos:

Corolario 4.3 En promedio, suponiendo que todas las permutaciones de las entradas son igualmente verosímiles, el número de comparaciones efectuadas por Quicksort (algoritmo 4.2) con conjuntos de tamaño n es aproximadamente $1.386\, n \lg n$, con n grande. □

⋆ Comportamiento promedio, con mayor exactitud

Aunque ya establecimos el comportamiento promedio de Quicksort, podemos aprender más volviendo a la ecuación de recurrencia (ecuación 4.1) y tratando de resolverla directamente para obtener algo más que sólo el primer término. En esta sección usaremos matemáticas algo avanzadas y el lector puede omitirla sin pérdida de continuidad.

Tenemos, por la ecuación (4.1),

$$A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i). \tag{4.2}$$

$$A(n-1) = n - 2 + \frac{2}{n-1} \sum_{i=1}^{n-2} A(i). \tag{4.3}$$

Si restamos la sumatoria de la ecuación (4.3) de la sumatoria de la ecuación (4.2), se cancelará la mayor parte de los términos. Puesto que las sumatorias se multiplican por factores distintos, necesitamos algo de álgebra un poco más complicada. Informalmente, calcularemos

$$n \times \text{ecuación (4.2)} - (n-1) \times \text{ecuación (4.3)}.$$

Entonces,

$$nA(n) - (n-1)\,A(n-1) = n(n-1) + 2\sum_{i=1}^{n-1} A(i) - (n-1)(n-2) - 2\sum_{i=1}^{n-2} A(i)$$

$$= 2\,A(n-1) + 2(n-1).$$

Así pues,

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

Ahora sea

$$B(n) = \frac{A(n)}{n+1}.$$

La ecuación de recurrencia para B es

$$B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)} \qquad B(1) = 0.$$

Con la ayuda de la ecuación (1.11), dejamos al lector verificar que

$$B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)}$$

$$\approx 2(\ln n + 0.577) - 4n/(n+1).$$

Por tanto,

$$A(n) \approx 1.386\, n \lg n - 2.846\, n.$$

Consumo de espacio

A primera vista podría parecer que Quicksort ordena en su lugar, pero no es así. Mientras el algoritmo está trabajando con un subintervalo, los índices inicial y final (llamémoslos fronteras) de todos los demás subintervalos que aún no se han ordenado se guardan en la pila de marcos, el tamaño de la pila dependerá del número de subintervalos en los que se dividirá el intervalo. Esto, desde luego, depende de n. En el peor caso, Partir separa un elemento a la vez; la profundidad de la recursión es n. Por tanto, la cantidad de espacio que la pila ocupa en el peor caso está en $\Theta(n)$. Una de las modificaciones del algoritmo que describiremos a continuación puede reducir considerablemente el tamaño máximo de la pila.

4.4.4 Mejoras al algoritmo de Quicksort básico

Selección del pivote

Hemos visto que Quicksort funciona bien si la clave `pivote` que Partir usa para dividir un segmento está cerca de la mitad del segmento. (Su posición es el valor `puntoPartir` que Partir devuelve.) Escoger `E[primero]` como pivote hace que Quicksort tenga un desempeño eficiente en los casos en que el ordenamiento debería ser fácil (por ejemplo, cuando el arreglo ya está ordenado). Hay varias otras estrategias para escoger el elemento pivote. Una consiste en escoger al azar un entero q entre `primero` y `ultimo` y hacer `pivote = E[q].clave`. Otra consiste en hacer que `pivote` sea la mediana de las claves de los elementos `E[primero]`, `E[(primero+ultimo)/2]` y `E[ultimo]`. (En ambos casos, el elemento que está en `E[primero]` se intercambiaría con el elemento pivote antes de proceder con el algoritmo Partir.) Ambas estrategias requieren

cierto trabajo extra para escoger `pivote`, pero lo compensan mejorando el tiempo de ejecución medio de un programa Quicksort.

Estrategia de partición alterna

La versión de Partir que presentamos en el texto es la que mueve menos elementos, en promedio, en comparación con otras estrategias de partición. La mostramos con subrutinas por claridad, si codificamos dichas subrutinas dentro del cuerpo de Partir, no como subrutinas aparte que se invocan, nos ahorraríamos algo de procesamiento fijo; sin embargo, algunos compiladores optimadores pueden efectuar este cambio automáticamente. En la sección de Notas y referencias al final del capítulo se mencionan otras consideraciones de optimación. En los ejercicios hay una versión alterna que es fácil de entender y programar, pero un poco más lenta.

Ordenamiento pequeño

Quicksort no es muy bueno para ordenar conjuntos pequeños, debido al procesamiento fijo que implica la invocación de procedimientos. Sin embargo, por la naturaleza del algoritmo, cuando n es grande Quicksort divide el conjunto en subconjuntos pequeños y los ordena recursivamente. Por tanto, siempre que el tamaño de un subconjunto es pequeño, el algoritmo se vuelve ineficiente. Este problema se puede remediar escogiendo un valor pequeño para `pequeño` y ordenando los subconjuntos cuyo tamaño sea menor o igual que `pequeño` con alguna técnica de ordenamiento sencilla, no recursiva, que llamaremos `smallSort` en el algoritmo modificado. (Ordenamiento por inserción es una buena opción.)

```
quickSort(elemento E, int primero, int ultimo)
   if(ultimo − primero < pequeño)
      elementoPivote = E[primero];
      pivote = elementoPivote.clave;
      int puntoPartir = partir(E, pivote, primero, ultimo);
      E[puntoPartir] = elementoPivote;
      quickSort(E, primero, puntoPartir−1);
      quickSort(E, puntoPartir + 1, ultimo);
   else
      smallSort(E, primero, ultimo);
```

Una variación de este tema consiste en omitir la invocación de `smallSort`. Entonces, cuando Quicksort termine, el arreglo no estará ordenado, pero ningún elemento se tendrá que mover más de `pequeño` posiciones para llegar a su posición correcta. (¿Por qué no?), por tanto, una sola ejecución de Ordenamiento por inserción como procesamiento posterior será muy eficiente y efectuará aproximadamente el mismo número de comparaciones que en todas sus invocaciones en su papel de `smallSort`.

¿Qué valor debe tener `pequeño`?, la mejor opción depende de la implementación específica del algoritmo (es decir, de la computadora empleada y de los detalles del programa), pues estamos haciendo algunas concesiones entre procesamiento fijo y comparaciones de claves. Un valor cercano a 10 deberá producir un desempeño razonable.

Optimación del espacio de pila

Observamos que la profundidad de recursión con Quicksort puede llegar a ser muy grande, proporcional a n en el peor caso (cuando Partir sólo separa un elemento en cada ocasión). Una bue-

na parte de las operaciones de apilar y desapilar que se efectuarán con la pila de marcos es innecesaria. Después de Partir, el programa comienza a ordenar el subintervalo `E[primero],...,E[puntoPartir − 1]`; después deberá ordenar el subintervalo `E[puntoPartir + 1],...,E[ultimo]`.

La segunda invocación recursiva es el último enunciado del procedimiento, por lo que se puede convertir en una iteración como ya vimos antes con `desplaVac` en Ordenamiento por inserción. Queda la primera invocación recursiva, así que sólo hemos eliminado parcialmente la recursión.

Aunque sólo queda una invocación recursiva en el procedimiento, de todos modos debemos tratar de evitar que la profundidad de recursión sea excesiva. Ello podría suceder después de una serie de invocaciones recursivas, cada una de las cuales trabaja con un subintervalo que apenas es más pequeño que el anterior. Por tanto, nuestro segundo ardid consiste en evitar hacer la invocación recursiva con el subintervalo *más grande*. Si garantizamos que cualquier invocación recursiva trabajará con cuando más la mitad de los elementos con que trabajó su invocación "progenitora", aseguraremos que la profundidad de recursión se mantenga por debajo de aproximadamente lg n. Combinamos estas dos ideas en la versión siguiente, en la que "ORT" significa "optimación de recursión trasera". De lo que se trata es que, después de cada partición, la siguiente invocación recursiva trabaje con el subintervalo más pequeño y que el mayor se maneje directamente en el ciclo **`while`**.

```
quickSortORT(Elemento[], int primero, int ultimo)
    int primero1, ultimo1, primero2, ultimo2;

    primero2 = primero; ultimo2 = ultimo;
    while (ultimo2 − primero2 > 1)
       elementoPivote = E[primero];
       pivote = elementoPivote.clave;
       int puntoPartir = partir(E, pivote, primero2, ultimo2);
       E[puntoPartir] = elementoPivote;
       if (puntoPartir < (primero2 + ultimo2) / 2)
            primero1 = primero2; ultimo1 = puntoPartir − 1;
            primero2 = puntoPartir + 1; ultimo2 = ultimo1;
       else
            primero1 = puntoPartir + 1; ultimo1 = ultimo2;
            primero2 = primero1; ultimo2 = puntoPartir − 1;
       quickSortORT(E, primero1, ultimo1);
       // Continuar el ciclo con primero2, ultimo2.
    return;
```

Mejoras combinadas

Hemos tratado de forma independiente las modificaciones anteriores, pero son compatibles y se pueden combinar en un mismo programa.

Comentarios

En la práctica, los programas Quicksort se ejecutan con gran rapidez en promedio cuando n es grande, y se les usa ampliamente. Sin embargo, en el peor caso Quicksort tiene un desempeño pobre. Al igual que Ordenamiento por inserción (sección 4.2), Maxsort y Ordenamiento de burbuja

(ejercicios 4.1 y 4.2), el tiempo de peor caso de Quicksort está en $\Theta(n^2)$ aunque, a diferencia de los otros, su comportamiento promedio está en $\Theta(n \log n)$. ¿Existen algoritmos de ordenamiento cuyo tiempo de peor caso esté en $\Theta(n \log n)$ o podemos establecer una cota inferior de peor caso de $\Theta(n^2)$? El enfoque de Divide y vencerás nos proporcionó la mejora en el comportamiento promedio. Examinemos otra vez la técnica general y veamos cómo podemos usarla para mejorar el comportamiento de peor caso.

4.5 Fusión de sucesiones ordenadas

En esta sección examinaremos una solución sencilla al problema siguiente: dadas dos sucesiones A y B en orden no decreciente, fusionarlas para crear una sucesión ordenada C. La fusión de subsucesiones ordenadas es fundamental para la estrategia de Mergesort, pero también tiene numerosas aplicaciones por derecho propio, algunas de las cuales se verán en los ejercicios. La medida del trabajo efectuado por un algoritmo de fusión será el número de comparaciones de claves que el algoritmo realiza.

Sean k y m el número de elementos de las sucesiones A y B, respectivamente. Sea $n = k + m$ el "tamaño del problema". Suponiendo que ni A ni B están vacías, podemos determinar de inmediato el primer elemento de C: es el mínimo del primer elemento de A y el primer elemento de B. ¿Y el resto de C?, supóngase que el primer elemento de A fue el mínimo. Entonces el resto de C deberá ser el resultado de fusionar todos los elementos de A *después* del primero, con todos los elementos de B. Sin embargo, ésta no es más que una versión más pequeña del problema original. La situación es simétrica si el primer elemento de B fue el mínimo. En ambos casos, el tamaño del problema restante (construir el resto de C) es $n - 1$. Ello nos recuerda el Método 99 (sección 3.2.2).

Si suponemos que sólo es necesario fusionar problemas cuyo tamaño sea cuando más 100 y podemos invocar `fusionar99` para fusionar problemas de tamaño 99 o menor, el problema ya está resuelto. He aquí el pseudocódigo:

```
fusionar(A, B, C)
    if (A está vacía)
        resto de C = resto de B
    else if (B está vacía)
        resto de C = resto de A
    else
        if (primero de A es menor que primero de B)
            primero de C = primero de A
            fusionar99(resto de A, B, resto de C)
        else
            primero de C = primero de B
            fusionar99(A, resto de B, resto de C)
    return
```

Ahora basta cambiar `fusionar99` a `fusionar` para tener la solución recursiva general.

Una vez que se percibe la idea de la solución, es evidente también cómo puede formularse una solución iterativa. La idea funciona con todas las estructuras de datos secuenciales, pero plan-

tearemos el algoritmo en términos de arreglos, a fin de hacerlo más definido. Introduciremos tres índices para mantenernos al tanto de dónde comienzan "resto de A", "resto de B" y "resto de C" en cualquier etapa de la iteración. (Estos índices serían parámetros en la versión recursiva.)

Algoritmo 4.4 Fusionar

Entradas: Arreglos A con k elementos y B con m elementos, ambos en orden no decreciente según sus claves.

Salidas: C, un arreglo que contiene $n = k + m$ elementos de A y B en orden no decreciente. C se pasa como parámetro de entrada y el algoritmo lo llena.

```
void fusionar(Elemento[] A, int k, Elemento[] B, int m, Elemento[] C)
    int n = k + m;
    int indiceA = 0, indiceB = 0, indiceC = 0;
    // indiceA es el principio del resto de A; lo mismo para B, C.

    while (indiceA < k && indiceB < m)
        if (A[indiceA].clave <= B[indiceB].clave)
            C[indiceC] = A[indiceA];
            indiceA ++;
            indiceC ++;
        else
            C[indiceC] = B[indiceB];
            indiceB ++;
            indiceC ++;
        // Continuar el ciclo
    if (indiceA >= k)
        Copiar B[indiceB,..., m-1] en C[indiceC,..., n-1].
    else
        Copiar A[indiceA,..., k-1] en C[indiceC,..., n-1].
```

4.5.1 Peor caso

Siempre que se efectúa una comparación de claves de A y B, al menos un elemento se coloca en C y nunca vuelve a examinarse. Después de la última comparación, al menos dos elementos los que se acaban de comparar no se han colocado aún en C. El menor se coloca de inmediato en C, pero ahora C tiene cuando más $n - 1$ elementos y no se efectuarán más comparaciones. Los elementos que queden en el otro arreglo se colocan en C sin efectuar más comparaciones. Entonces, se efectúan cuando más $n - 1$ comparaciones. El peor caso necesitar las $n - 1$ comparaciones se presenta cuando $A[k - 1]$ y $B[m - 1]$ van en las dos últimas dos posiciones de C.

4.5.2 Optimidad de la fusión

Ahora demostraremos que el algoritmo 4.4 es óptimo en el peor caso entre los algoritmos basados en comparaciones cuando $k = m = n/2$. Es decir, para cualquier algoritmo basado en comparaciones que fusiona correctamente todas las entradas para las cuales $k = m = n/2$ debe haber

alguna entrada con la que es preciso efectuar $n - 1$ comparaciones. (Esto no quiere decir que con una entrada *específica* ningún algoritmo podría funcionar mejor que el algoritmo 4.4.) Después de considerar $k = m = n/2$, examinaremos algunas otras relaciones entre k y m.

Teorema 4.4 Cualquier algoritmo para fusionar dos arreglos ordenados, cada uno de los cuales contiene $k = m = n/2$ elementos, por comparación de claves, efectúa al menos $n - 1$ comparaciones de claves en el peor caso.

Demostración Supóngase que se nos da un algoritmo de fusión arbitrario. Sean a_i y b_i los i-ésimos elementos de A y B respectivamente. Demostraremos que es posible escoger claves tales que el algoritmo deba comparar a_i con b_i para $0 \leq i < m$, y a_i con b_{i+1} para $0 \leq i < m - 1$. Específicamente, escogemos claves tales que, siempre que el algoritmo compare a_i con b_i si $i < j$, el resultado es que $a_i < b_j$, y si $i \geq j$, el resultado es que $b_j < a_i$. Si escogemos las claves de modo tal que

$$b_0 < a_0 < b_1 < a_1 < \cdots < b_i < a_i < b_{i+1} < \cdots < b_{m-1} < a_{m-1} \tag{4.4}$$

se cumplan esas condiciones. No obstante, si para alguna i el algoritmo nunca compara a_i con b_i, bastará escoger claves en el mismo orden que en la ecuación (4.4), con la excepción de que $a_i < b_i$, para satisfacer también esas condiciones y el algoritmo no podría determinar el orden correcto.

Asimismo, si para alguna i el algoritmo nunca compara a_i con b_{i+1}, el acomodo de la ecuación (4.4) con la excepción de que $b_{i+1} < a_i$ sería congruente con los resultados de las comparaciones efectuadas y otra vez el algoritmo no podría determinar el orden correcto. □

¿Podemos generalizar esta conclusión? Supóngase que k y m son apenas diferentes (como veremos que podrían ser en Mergesort).

Corolario 4.5 Cualquier algoritmo para fusionar dos arreglos ordenados comparando claves, donde las entradas contienen k y m elementos, respectivamente, k y m difieren en 1, y $n = k + m$, efectúa por lo menos $n - 1$ comparaciones de claves en el peor caso.

Demostración Es válida la demostración del teorema 4.4, excepto que no hay a_{m-1}. □

¿Podemos generalizar aún más esta conclusión? Si encontramos un tipo de comportamiento en un extremo, suele ser recomendable verificar el otro extremo. Aquí, el primer "extremo" era $k = m$, así que en el otro extremo k y m serán tan diferentes como sea posible. Examinemos un caso extremo, en el que $k = 1$ y m es grande, de modo que $n = m + 1$. Podemos idear un algoritmo que efectúe cuando más $\lceil \lg(m + 1) \rceil$ comparaciones. (¿Cuál es?) Entonces, es evidente que $n - 1$ no es una cota inferior en este caso. La mejora para $k = 1$ se puede generalizar a otros casos en los que k es mucho menor que n (véase el ejercicio 4.24). Por tanto, los argumentos de cota inferior del teorema 4.4 y el corolario 4.5 no se pueden extender a todas las combinaciones de k y m. Si desea conocer más posibilidades, vea el ejercicio 4.33 después de leer la sección 4.7.

4.5.3 Consumo de espacio

Por la forma en que está escrito el algoritmo 4.4, podría parecer que la fusión de sucesiones con un total de n elementos requiere suficientes posiciones de memoria para $2n$ elementos, ya que to-

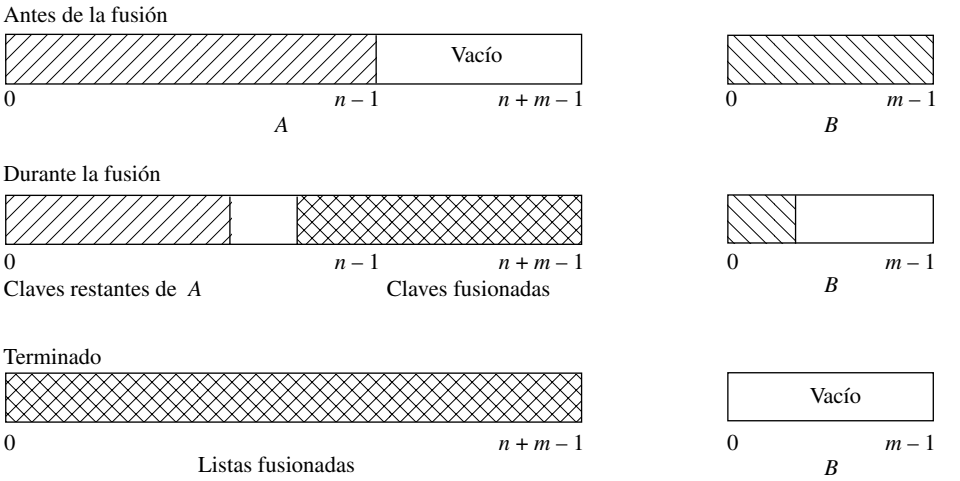


Figura 4.12 Arreglos traslapantes que se fusionan

dos los elementos se copian en *C*. En algunos casos, empero, se puede reducir la cantidad de espacio adicional necesario. Un caso así sería que las sucesiones sean listas ligadas, y *A* y *B* ya no se vayan a necesitar (como listas) una vez terminada la fusión. En tal caso los nodos de las listas *A* y *B* se podrán reciclar conforme se crea *C*.

Supóngase que las sucesiones de entrada están almacenadas en arreglos y que $k \geq m$. Si *A* tiene suficiente espacio para $n = k + m$ elementos, entonces sólo se necesitarán las *m* posiciones extra de *A*. Basta con identificar *C* con *A* y efectuar la fusión a partir de los extremos derechos (claves más grandes) de *A* y *B*, como se indica en la figura 4.12. Los primeros *m* elementos trasladados a "*C*" ocuparán las posiciones extra de *A*. De ahí en adelante, se usarán las posiciones de *A* que se vayan desocupando. Siempre habrá una brecha (es decir, algunas posiciones vacías) entre el extremo de la porción fusionada del arreglo y los elementos restantes de *A*, hasta que se hayan fusionado todos los elementos. Observe que si se emplea esta organización de almacenamiento para ahorrar espacio, se podrán eliminar las últimas líneas del algoritmo de fusión (**`else`** Copiar `A[indiceA],..., A[k-1]` en `C[indiceC],..., C[n-1]`) porque, si *B* se vacía antes que *A*, los elementos restantes de *A* ya estarán en su posición correcta y no será necesario cambiarlos de lugar.

Sea que *C* traslape o no uno de los arreglos de entrada, el espacio extra empleado por el algoritmo Fusionar cuando $k = m = n/2$ está en $\Theta(n)$.

4.6 Mergesort

El problema que presenta Quicksort es que Partir no siempre divide el arreglo en dos subintervalos iguales. Mergesort simplemente parte el arreglo en dos mitades y las ordena por separado (y naturalmente, de forma recursiva). Luego se fusionan las mitades ordenadas (véase la figura 4.13).

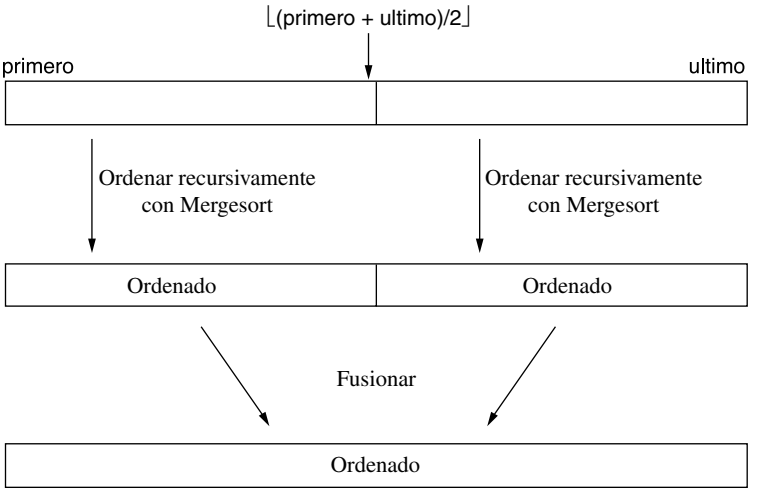


Figura 4.13 Estrategia de Mergesort

Así, empleando la terminología de divide y vencerás de la sección 4.3, `dividir` se limita a calcular el índice medio del subintervalo y no compara claves; `combinar` se encarga de la fusión.

Suponemos que Fusionar se modifica de modo que fusione subintervalos adyacentes de un arreglo, colocando el arreglo fusionado resultante de vuelta en las celdas que originalmente ocupaban los elementos que se fusionaron. Ahora sus parámetros son el nombre del arreglo *E*, los índices `primero`, `medio` y `ultimo` de los subintervalos que va a fusionar; es decir, los subintervalos ordenados son `E[primero]`,..., `E[medio]` y `E[medio+1]`,..., `E[ultimo]`, y el intervalo final ordenado será `E[primero]`,..., `E[ultimo]`. En esta modificación, la subrutina `fusionar` también se encarga de asignar el espacio de trabajo adicional requerido. Algunos aspectos de esto se trataron en la sección 4.5.3.

Algoritmo 4.5 Mergesort

Entradas: Arreglo *E* e índices `primero` y `ultimo`, tales que estén definidos los elementos de `E[i]` para `primero` $\leq i \leq$ `ultimo`.

Salidas: `E[primero]`,..., `E[ultimo]` es un reacomodo ordenado de los mismos elementos.

```
void mergeSort(Elemento[] E, int primero, int ultimo)
   if(primero < ultimo)
      int medio = (primero+ultimo) / 2;
      mergeSort(E, primero, medio);
      mergeSort(E, medio + 1, ultimo);
      fusionar(E, primero, medio, ultimo);
   return;
```

Hemos observado que los estudiantes a menudo confunden los algoritmos Fusionar y Mergesort. Recordemos que Merge*sort* es un algoritmo de ordenamiento; toma *un* arreglo desordenado y lo ordena. Fusionar toma *dos* arreglos que ya están ordenados y los combina para formar un arreglo ordenado.

Análisis de Mergesort

Primero, determinamos el orden asintótico del número de comparaciones de claves que Mergesort efectúa en el peor caso. Como siempre, definimos el tamaño del problema como $n = \texttt{ultimo} - \texttt{primero} + 1$, el número de elementos que hay en el intervalo a ordenar. La ecuación de recurrencia para el comportamiento de peor caso de Mergesort es

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \qquad (4.5)$$
$$W(1) = 0.$$

El Teorema maestro nos dice de inmediato que $W(n) \in \Theta(n \log n)$, así que por fin tenemos un algoritmo de ordenamiento cuyo comportamiento de peor caso está en $\Theta(n \log n)$. En vez de efectuar un análisis aparte para determinar la complejidad promedio de Mergesort, aplazaremos esta cuestión hasta haber desarrollado el teorema 4.11 acerca del comportamiento promedio, que es muy general, en la sección 4.7, que sigue a continuación.

Una posible desventaja de Mergesort es el espacio de trabajo auxiliar que requiere. Debido al espacio extra empleado para la fusión, que está en $\Theta(n)$, Mergesort no es un ordenamiento en su lugar.

⋆ Análisis de Mergesort, con más exactitud

Tiene cierto interés obtener una estimación más exacta del número de comparaciones en el peor caso, en vista de las cotas inferiores que se desarrollarán en la sección siguiente (sección 4.7). Veremos que Mergesort se acerca mucho a la cota inferior. Los lectores pueden pasar por alto los detalles de esta sección sin pérdida de continuidad y pasar a su conclusión principal, el teorema 4.6.

En el árbol de recursión de la ecuación (4.5) (véase la figura 4.14), observamos que la sumatoria de los costos no recursivos de los nodos con profundidad d es $n - 2^d$ (para todos los niveles del árbol que no contienen casos base). Podemos determinar que todos los casos base (para los que $W(1) = 0$) se presentan en las profundidades $\lceil \lg(n + 1) \rceil - 1$ o $\lceil \lg(n + 1) \rceil$. Hay exactamente n nodos de caso base. Sea la profundidad máxima D (es decir, $D = \lceil \lg(n + 1) \rceil$) y sea B el número de casos base en la profundidad $D - 1$. Entonces habrá $n - B$ casos base en la profundidad D (y ningún otro nodo en la profundidad D). Cada nodo no base en la profundidad $D - 1$ tiene dos hijos, de modo que hay $(n - B)/2$ casos no base en la profundidad $D - 1$. Utilizando esta información, calculamos la sumatoria de los costos no recursivos para los niveles más bajos del árbol así:

1. A la profundidad $D - 2$ hay 2^{D-2} nodos, ninguno de los cuales son casos base. La sumatoria de los costos no recursivos para este nivel es $n - 2^{D-2}$.
2. A la profundidad $D - 1$ hay $(n - B)/2$ nodos de casos no base. Cada uno tiene un problema de tamaño 2 (con costo 1), así que la sumatoria de los costos no recursivos para este nivel es $(n - B)/2$.
3. A la profundidad D hay $n - B$ casos base, con costo 0.

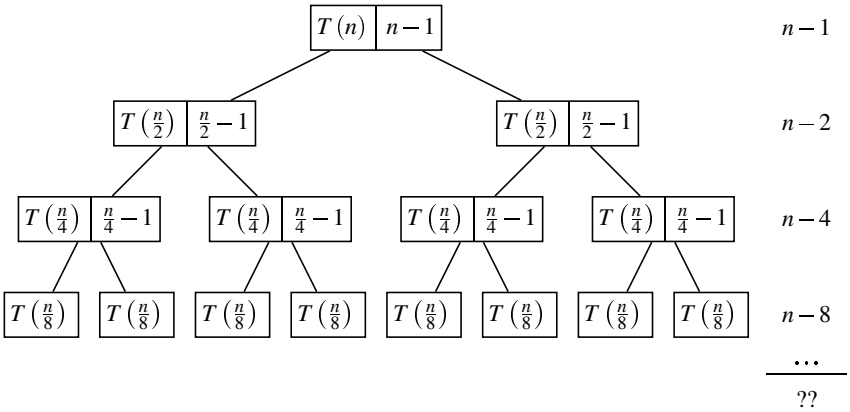


Figura 4.14 Árbol de recursión para Mergesort. Siempre que el parámetro de tamaño de nodo es impar, el tamaño del hijo izquierdo se redondea hacia arriba y el tamaño del hijo derecho se redondea hacia abajo (de dos y más líneas con punto).

El lector puede verificar que $B = 2^D - n$ (ejercicio 4.29). Por tanto

$$\begin{aligned} W(n) &= \sum_{d=0}^{D-2} (n - 2^d) + (n - B)/2 \\ &= n(D - 1) - 2^{D-1} + 1 + (n - B)/2 \\ &= n\,D - 2^D + 1. \end{aligned} \tag{4.6}$$

Puesto que D se redondea al entero mayor más cercano y aparece en el exponente, es difícil saber cómo se comporta la ecuación (4.6) entre potencias de 2. Demostraremos el teorema siguiente, que elimina la función techo del exponente.

Teorema 4.6 El número de comparaciones efectuadas por Mergesort en el peor caso está entre $\lceil n \lg(n) - n + 1 \rceil$ y $\lceil n \lg(n) - .914\ n \rceil$.

Demostración Si definimos $\alpha = 2^D/n$, entonces $1 \leq \alpha < 2$, y podremos sustituir a D en todas las partes de la ecuación (4.6) por $(\lg(n) + \lg(\alpha))$. Esto nos lleva a $W(n) = n \lg(n) - (\alpha - \lg \alpha)n + 1$. El valor mínimo de $(\alpha - \lg \alpha)$ es de aproximadamente .914 (véase el ejercicio 4.30) y el máximo dentro del intervalo considerado es 1. □

Así pues, Mergesort efectúa aproximadamente 30% menos comparaciones en el peor caso que las que Quicksort efectúa en promedio. Por otra parte, Mergesort cambia de lugar más elementos que Quicksort en promedio, por lo que es posible que no sea más rápido (véanse los ejercicios 4.21 y 4.27).

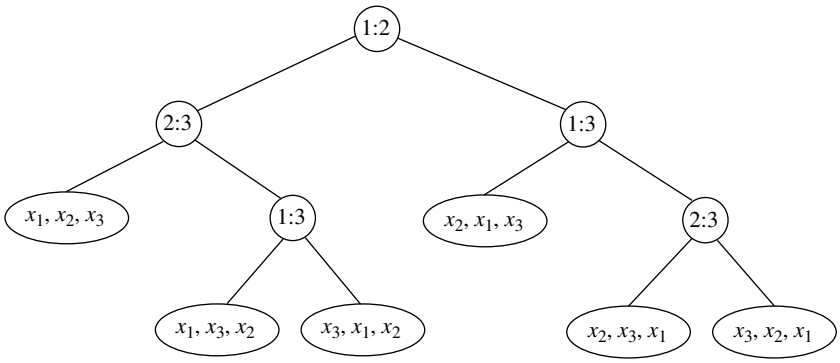


Figura 4.15 Árbol de decisión para un algoritmo de ordenamiento, $n = 3$

4.7 Cotas inferiores para ordenar comparando claves

El número de comparaciones de claves que efectúan Ordenamiento por inserción y Quicksort en el peor caso está en $\Theta(n^2)$. Pudimos mejorar esto con Mergesort, cuyo peor caso está en $\Theta(n \log n)$. ¿Podemos mejorarlo aún más?

En esta sección deduciremos cotas inferiores para el número de comparaciones que debe efectuar en el peor caso y en promedio cualquier algoritmo que ordene comparando claves. Estos resultados nos dicen cuándo podemos dejar de buscar un mejor algoritmo. Para deducir las cotas inferiores suponemos que todas las claves del arreglo a ordenar son distintas.

4.7.1 Árboles de decisión para algoritmos de ordenamiento

Sea n fijo y supóngase que las claves son $x_1, x_2, \ldots, x_n$. Asociaremos a cada algoritmo y entero positivo n un árbol de decisión (binario) que describe la sucesión de comparaciones efectuadas por el algoritmo con cualquier entrada de tamaño n. Sea Ordenar cualquier algoritmo que ordena comparando claves. Cada comparación tiene una ramificación de dos vías (puesto que las claves son distintas), y suponemos que Ordenar tiene una instrucción de salida que genera el arreglo reacomodado de claves. El árbol de decisión de Ordenar se define inductivamente asociando un árbol a cada comparación y cada instrucción de salida como sigue. El árbol asociado a una instrucción de salida consiste en un nodo rotulado con el reacomodo de las claves. El árbol asociado a una instrucción que compara las claves x_i y x_j consiste en una raíz rotulada $(i : j)$, un subárbol izquierdo que es el árbol asociado a la instrucción (de comparación o salida) que se ejecutará a continuación si $x_i < x_j$ y un subárbol derecho que es el árbol asociado a la instrucción (de comparación o salida) que se ejecutará a continuación si $x_i > x_j$. El árbol de decisión para Ordenar es el árbol asociado a la primera instrucción de comparación que ejecuta. La figura 4.15 muestra un ejemplo de árbol de decisión para $n = 3$.

La acción de Ordenar con una entrada específica corresponde a seguir un camino de su árbol de decisión desde la raíz hasta una hoja. El árbol debe tener por lo menos $n!$ hojas, porque hay $n!$ formas de permutar las claves. Puesto que el camino único que se sigue con cada entrada depende únicamente del ordenamiento de las claves y no de sus valores específicos, es posible llegar a exactamente $n!$ hojas desde la raíz ejecutando realmente Ordenar. Supondremos que se eliminan

cualesquier caminos del árbol que nunca se siguen. También supondremos que los nodos de comparación que sólo tienen un hijo se eliminan y son reemplazados por el hijo, y que esta "poda" se repite hasta que todos los nodos internos tienen grado 2. El árbol podado representa un algoritmo que es por lo menos tan eficiente como el original, así que las cotas inferiores que deduzcamos empleando árboles con exactamente $n!$ hojas y nodos internos todos de grado 2 serán cotas inferiores válidas para todos los algoritmos que ordenan comparando claves. De aquí en adelante supondremos que Ordenar se describe con un árbol de este tipo.

El número de comparaciones efectuadas por Ordenar con una entrada específica es el número de nodos internos que están en el camino que se sigue con esa entrada. Por tanto, el número de comparaciones efectuadas en el peor caso es el número de nodos internos que hay en el camino más largo y ésa es la altura del árbol. El número promedio de comparaciones efectuadas es la media de las longitudes de todos los caminos desde la raíz hasta una hoja. (Por ejemplo, para $n = 3$, el algoritmo cuyo árbol de decisión se muestra en la figura 4.15 efectúa tres comparaciones en el peor caso y dos dos tercios en promedio.)

4.7.2 Cota inferior para el peor caso

Para obtener una cota inferior de peor caso al ordenar por comparación, deducimos una cota inferior para la altura de un árbol binario en términos el número de hojas, puesto que la única información cuantitativa que tenemos acerca de los árboles de decisión es el número de hojas.

Lema 4.7 Sea L el número de hojas de un árbol binario y sea h su altura. Entonces $L \leq 2^h$.

Demostración Una inducción directa con h. □

Lema 4.8 Sean L y h igual que en el lema 4.7. Entonces $h \geq \lceil \lg L \rceil$.

Demostración Si sacamos el logaritmo de ambos lados de la desigualdad del lema 4.7 obtenemos $\lg L \leq h$. Puesto que h es entero, $h \geq \lceil \lg L \rceil$. □

Lema 4.9 Para una n dada, el árbol de decisión de cualquier algoritmo que ordena comparando claves tiene una altura de por lo meno $\lceil \lg n! \rceil$.

Demostración Sea $L = n!$ en el lema 4.8. □

Así pues, el número de comparaciones necesarias para ordenar en el peor caso es de por lo menos $\lceil \lg n! \rceil$. Nuestro mejor ordenamiento hasta ahora es Mergesort, pero ¿qué tan cercano es $\lceil \lg n! \rceil$ a $n \lg n$? Para obtener la respuesta, necesitamos expresar $\lg n!$ en una forma más conveniente y determinar una cota inferior para su valor. Hay varias formas de hacerlo. Tal vez la forma más sencilla, aunque no muy exacta, de hacerlo es observar que

$$n! \geq n(n-1) \cdots (\lceil n/2 \rceil) \geq \left(\frac{n}{2}\right)^{\frac{n}{2}},$$

así que

$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2},$$

que está en $\Theta(n \log n)$. Por tanto, vemos ya que Mergesort tiene un orden asintótico óptimo. Para obtener una cota inferior más exacta, aprovechamos el hecho de que

$$\lg n! = \sum_{j=1}^{n} \lg(j).$$

Utilizando la ecuación (1.18), obtenemos

$$\lg n! \geq n \lg n - (\lg e)n,$$

donde e denota la base de los logaritmos naturales y $\lg(e)$ es aproximadamente 1.443. Así pues, la altura del árbol de decisión es de por lo menos $\lceil n \lg n - 1.443n \rceil$.

Teorema 4.10 Cualquier algoritmo para ordenar n elementos comparando claves debe efectuar por lo menos $\lceil \lg n! \rceil$, o aproximadamente $\lceil n \lg n - 1.443n \rceil$ comparaciones de claves en el peor caso. □

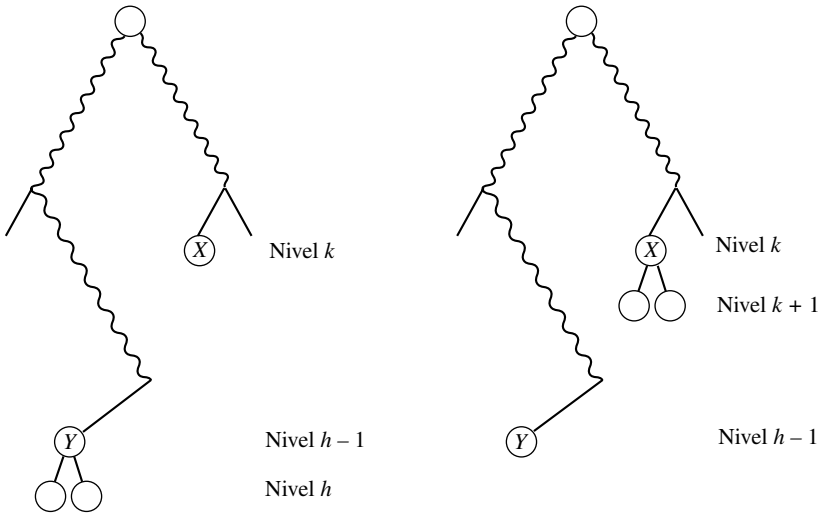
Así que Mergesort se acerca mucho a la optimidad. Existe cierta diferencia entre el comportamiento exacto de Mergesort y la cota inferior. Consideremos el caso en el que $n = 5$. El ordenamiento por inserción efectúa 10 comparaciones en el peor caso y Mergesort realiza 8, pero la cota inferior es $\lceil \lg 5! \rceil = \lceil \lg 120 \rceil = 7$. ¿Es simplemente que la cota inferior no es lo bastante buena o podemos encontrar un algoritmo mejor que Mergesort? Animamos al lector a tratar de encontrar una forma de ordenar cinco elementos con sólo siete comparaciones de claves en el peor caso (ejercicio 4.32).

4.7.3 Cota inferior del comportamiento promedio

Necesitamos una cota inferior para la media de las longitudes de todos los caminos desde la raíz hasta una hoja en un árbol de decisión. Recordemos la definición 3.2, que dice que un árbol binario en el que todos los nodos tienen grado 0 o 2 es un *árbol-2*. Las hojas de un árbol así pueden ser *nodos externos*, que son de tipo distinto que los nodos internos. Nuestros árboles de decisión son árboles-2 y todas sus hojas son instrucciones de salida, mientras que todos los nodos internos son instrucciones de comparación.

Recordemos la definición 3.3, según la cual la *longitud de camino externo* de un árbol es la suma de las longitudes de todos los caminos desde la raíz hasta un nodo externo (es decir, instrucción de salida); lo denotaremos con *lce*. Si un árbol de decisión tiene L hojas, la longitud media de los caminos desde la raíz hasta una hoja será lce/L.

Estamos buscando una cota inferior para *lce* de entre todos los árboles de decisión (árboles-2) que tienen L hojas, tomando L como fija por el momento. Podemos argumentar que los árboles (con L hojas) que minimizan *lce* están lo más equilibrados posible. Supóngase que tenemos un árbol-2 con altura h que tiene una hoja X a la profundidad k, donde k es dos o más menor que h. En la figura 4.16(a) se presenta una ilustración. La figura 4.16(b) muestra un árbol-2 con el mismo número de hojas y una *lce* más baja. Escogemos un nodo Y con profundidad $h - 1$ que no es una hoja, eliminamos sus dos hijos y conectamos esos dos hijos a X. El número total de hojas no ha



(a) Árbol-2 dado con L hojas

(b) Árbol-2 modificado con L hojas y longitud de camino externo reducida en $h - k - 1$

Figura 4.16 Reducción de la longitud de camino externo

cambiado, pero la *lce* sí. Tres caminos del árbol original (los caminos a los hijos de Y y el camino a X), cuyas longitudes son en total $h + h + k$, ya no se cuentan. Hay tres caminos nuevos (a Y y a los nuevos hijos de X) cuyas longitudes suman $h - 1 + 2(k + 1)$. El cambio neto en *lce* es $k + 1 - h$, que es negativo, así que la *lce* ha disminuido. Por consiguiente, si un árbol-2 tiene una *lce* mínima de entre todos los árboles-2 que tienen L hojas, su longitud de camino externo es de aproximadamente $L \lg(L)$.

El lema 3.7 hace más precisa esta cota; dice (en nuestro contexto) que cualquier árbol de decisión con L hojas tiene $lce \geq L \lg(L)$, así que la longitud media del camino a un nodo de instrucción de salida es por lo menos $\lg(L)$. Esto implica de forma inmediata el teorema siguiente:

Teorema 4.11 El número medio de comparaciones efectuadas por un algoritmo para ordenar n elementos mediante comparación de claves es por lo menos $\lg(n!)$, que es aproximadamente $n \lg n - 1.443n$. □

La única diferencia respecto a la cota inferior del peor caso es que no se redondea al entero superior más cercano; el promedio no necesita ser un entero, aunque el peor caso sí. Si bien nunca analizamos el comportamiento promedio de Mergesort, esta cota general nos permite concluir que no puede ser mucho más bajo que su peor caso; los términos iniciales deben coincidir y sólo hay una brecha de aproximadamente $0.5\ n$ en el término de segundo orden. Además, el caso promedio de Quicksort sólo puede mejorarse en un 30% cuando más, por más mejoras que se le hagan, como escoger el elemento de partición con más cuidado.

4.8 Heapsort

Quicksort reacomoda los elementos en el arreglo original, pero no puede garantizar que hará una subdivisión pareja del problema, por lo que su peor caso es muy malo. Mergesort puede garantizar una subdivisión pareja y tiene un peor caso casi óptimo, pero no puede reacomodar los elementos en el arreglo original; necesita un espacio de trabajo auxiliar considerable. Heapsort reacomoda los elementos en el arreglo original y su peor caso está en $\Theta(n \log n)$, que es óptimo en términos de tasa de crecimiento, así que en cierto sentido combina las ventajas de Quicksort y Mergesort. La desventaja de Heapsort es un factor constante más alto que el de los otros dos. Sin embargo, una versión más nueva de Heapsort reduce este factor constante a un nivel tal que puede competir con Quicksort y Mergesort. Llamamos a esta versión más nueva *Heapsort Acelerado*. Por lo anterior, Heapsort Acelerado podría convertirse en el método de ordenamiento preferido.

4.8.1 Montones

El algoritmo Heapsort emplea una estructura de datos llamada *montón* (*heap*), que es un árbol binario con algunas propiedades especiales. La definición de un montón incluye una descripción de la estructura y una condición que deben satisfacer los datos de los nodos, llamada *propiedad de árbol en orden parcial*. Informalmente, una *estructura* de montón es un árbol binario completo del que se han eliminado algunas hojas de extrema derecha. (En la figura 4.17 se presentan ilustraciones.) Un montón permite implementar de forma eficiente el tipo de datos abstracto de *cola de prioridad* (sección 2.5.1). En un montón, el elemento de prioridad "más alta" se guarda en la raíz del árbol binario. Dependiendo de la noción de prioridad, este elemento podría tener la clave más baja (en el caso de un montón minimizante). Para que Heapsort ordene en forma ascendente, se usa un montón maximizante, así que describiremos los montones en estos términos. En otros casos utilizaremos montones minimizantes.

Usaremos la terminología de que S es un conjunto de elementos cuyas claves tienen un ordenamiento lineal, y T es un árbol binario de altura h cuyos nodos contienen elementos de S.

Definición 4.1 Estructura de montón

Un árbol binario T es una *estructura* de montón si y sólo si satisface las condiciones siguientes:

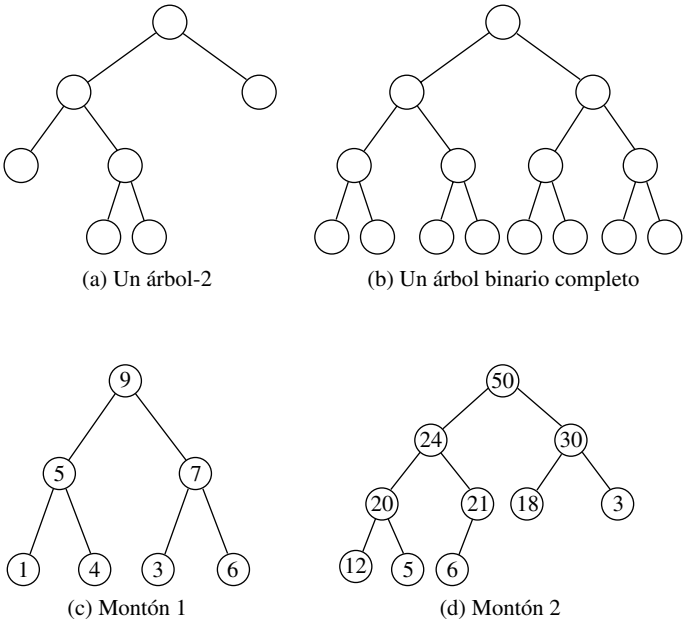
1. T está completo al menos hasta la profundidad $h - 1$.
2. Todas las hojas están a una profundidad h o $h - 1$.
3. Todos los caminos a una hoja de profundidad h están a la izquierda de todos los caminos a una hoja de profundidad $h - 1$.

El nodo interno de la extrema derecha a la profundidad $h - 1$ en una estructura de montón puede tener sólo su hijo izquierdo (pero no sólo el hijo derecho). Todos los demás nodos internos tienen dos hijos. La estructura de montón también lleva el nombre de árbol binario *completo a la izquierda*. ■

Definición 4.2 Propiedad de árbol en orden parcial

Un árbol T es un *arbol en orden parcial* (maximizante) si y sólo si la clave en cualquier nodo es mayor o igual que las claves en cada uno de sus hijos (si tiene alguno). ■

Observe que un árbol binario completo es una estructura de montón. Si se añaden nodos nuevos a un montón, se deben agregar de izquierda a derecha en el nivel más bajo y si se elimina un



(a) Un árbol-2 (b) Un árbol binario completo

(c) Montón 1 (d) Montón 2

Figura 4.17 Árboles-2, árboles binarios completos y montones

nodo, deberá ser el de la extrema derecha en el nivel más bajo si se desea que la estructura resultante siga siendo un montón. Observe que la raíz debe contener la clave más grande del montículo.

4.8.2 La estrategia de Heapsort

Si los elementos a ordenar están acomodados en un montón, podremos construir una sucesión ordenada en orden inverso eliminando repetidamente el elemento de la raíz (la clave más grande restante) y reacomodando los elementos que quedan en el montón a modo de restablecer la propiedad de árbol en orden parcial, lo que llevará la siguiente clave más grande a la raíz. Esta operación no es sino `borrarMax` del TDA de cola de prioridad. (Podríamos construir la sucesión ordenada en orden ascendente con un montón minimizante; la razón por la que usamos un montón maximizante quedará clara cuando estudiemos una implementación especialmente eficiente en la sección 4.8.5.)

Puesto que este enfoque requiere construir primero un montón y luego ejecutar repetidamente `borrarMax`, lo que implica cierto reacomodo de los elementos del montón, no parece ser una estrategia prometedora para llegar a un algoritmo de ordenamiento eficiente. Pese a ello, su desempeño es muy bueno. Delinearemos la estrategia aquí y luego precisaremos los detalles. Como siempre, supondremos que los n elementos están almacenados en un arreglo `E`, pero en esta ocasión supondremos que el intervalo de índices es $1, \ldots, n$, por razones que se harán evidentes cuando examinemos la implementación del montón. Por el momento, supondremos que el montón (llamado H) está en otro lado.

```
heapSort(E, n) // BOSQUEJO
    Construir H a partir de E, el conjunto de n elementos a ordenar;
    for(i = n; i ≥ 1; i --)
        maxActual = obtMax(H);
        borrarMax(H);
        E[i] = maxActual;
```

El primer y último diagramas de la figura 4.18 muestran un ejemplo antes y después de una iteración del ciclo **for**. Los diagramas intermedios muestran pasos de los reacomodos que efectúan `borrarMax` y la subrutina `repararMonton`, invocada por `borrarMax`.

```
borrarMax(H) // BOSQUEJO
    Copiar en K el elemento de extrema derecha del nivel más bajo de H.
    Borrar el elemento de extrema derecha del nivel más bajo de H.
    repararMonton(H, K);
```

Como puede verse, casi todo el trabajo corre por cuenta de `repararMonton`.

Ahora necesitamos un algoritmo para construir un montón y un algoritmo para `repararMonton`. Puesto que `repararMonton` puede servir para resolver también el problema de construir un montón, la examinaremos a continuación.

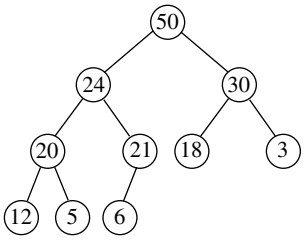
4.8.3 Reparar montón

El procedimiento `repararMonton` restaura la propiedad de árbol en orden parcial en una estructura de montón en la que esa propiedad ya existe en todas sus partes con la posible excepción de la raíz. En términos específicos, cuando inicia `repararMonton`, tenemos una estructura de montón con una raíz "vacante". Los dos subárboles son árboles en orden parcial, y tenemos un elemento adicional, digamos *K*, que insertar. Puesto que la raíz está vacante, comenzaremos ahí y dejaremos que *K* y el nodo vacante se filtren hacia abajo a sus posiciones correctas. En su posición final, *K* (o, más bien, la clave de *K*) deberá ser mayor o igual que todos sus hijos, así que en cada paso *K* se compara con el mayor de los hijos del nodo que actualmente está vacante. Si *K* es mayor (o igual), podrá insertarse en el nodo vacante; si no, el hijo mayor subirá al nodo vacante y se repetirá el proceso.

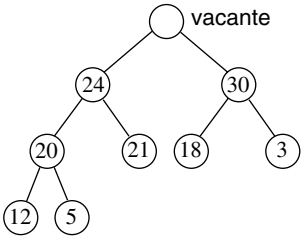
Ejemplo 4.1 RepararMontón en acción

La acción de `repararMonton` se ilustra en los diagramas segundo a quinto de la figura 4.18. Retrocediendo un poco, el primer diagrama muestra la configuración inicial, al principio del ciclo **for** del bosquejo de `heapSort` que presentamos en la sección 4.8.2. Primero, `heapSort` copia en `maxActual` la clave 50 de la raíz de *H*, con lo que la raíz del árbol queda de hecho vacante; luego invoca `borrarMax`, que copia en *K* la clave 6 del nodo de extrema derecha del nivel inferior del árbol y elimina ese nodo.

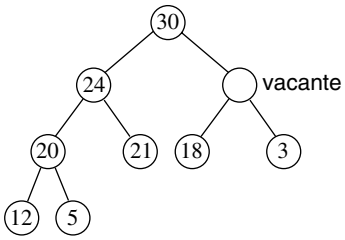
Esto nos lleva al segundo diagrama, y es ahí donde comienza a trabajar `repararMonton(H, K)`. El hijo mayor del nodo vacante es 30 y también es mayor que *K* (que es 6), así que 30 sube a la posición vacante y el nodo vacante se filtra hacia abajo, lo que lleva al tercer diagrama. Una vez más, el hijo mayor del nodo vacante es mayor que *K*, así que el nodo vacante se filtra otra vez hacia abajo. Ahora el nodo vacante es una hoja, así que podemos insertar a *K* ahí y habremos restaurado la propiedad de árbol en orden parcial de *H*. ■



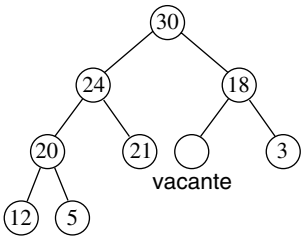
(a) El montón



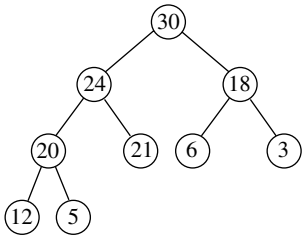
(b) Se ha sacado la clave que estaba en la raíz; se ha quitado la hoja de extrema derecha del nivel más bajo. Es preciso reinsertar $K = 6$.



(c) El hijo mayor de vacante, 30, es mayor que K, así que sube y vacante baja.



(d) El hijo mayor de vacante, 18, es mayor que K, así que sube y vacante baja.



(e) Por último, puesto que vacante es una hoja, insertamos $K = 6$.

Figura 4.18 Eliminación del elemento que está en la raíz y restablecimiento de la propiedad de árbol en orden parcial.

Aunque la estructura de árbol del montón es indispensable para justificar y entender Heapsort, veremos más adelante que es posible representar montones y submontones sin aristas explícitas.

Algoritmo 4.6 RepararMontón (bosquejo)

Entradas: Un árbol binario no vacío *H* con una raíz "vacante", tal que sus subárboles izquierdo y derecho sean árboles en orden parcial y un elemento `K` a insertar. El tipo de *H* se llamará `Heap` en este bosquejo. Suponemos que los nodos de *H* son de tipo `Elemento`.

Salidas: Un árbol binario *H* que consiste en `K` y los elementos originales de *H* y satisface la propiedad de árbol en orden parcial.

Comentario: La estructura de *H* no se altera, pero sí cambia el contenido de sus nodos.

```
repararMonton(H, K) // BOSQUEJO
   if (H es una hoja)
       insertar K en raiz(H);
   else
       asignar subarbolIzq(H) o subarbolDer(H) a subMontonMayor, el que tenga
       la clave más grande en su raíz. Esto implica una comparación de claves, a menos que
       subarbolDer esté vacío.
       if (K.clave ≥ raiz(subMontonMayor).clave)
          insertar K en raiz(H);
       else
          insertar raiz(subMontonMayor) en raiz(H);
          repararMonton(subMontonMayor, K);
   return;
```

Lema 4.12 El procedimiento `repararMonton` efectúa $2h$ comparaciones de claves en el peor caso, con un montón de altura h.

Demostración Se efectúan cuando más dos comparaciones de claves en cada activación del procedimiento y la altura del árbol se reduce en uno durante la invocación recursiva. (Una comparación está implícita en la determinación de `subMontonMayor`.) □

4.8.4 Construcción de montones

Supóngase que inicialmente colocamos todos los elementos en una estructura de montón en orden arbitrario; es decir, no necesariamente se satisface la propiedad de árbol en orden parcial en cualquier submontón. El algoritmo `repararMonton` sugiere un enfoque de Divide y vencerás para establecer la propiedad de árbol en orden parcial. Los dos subárboles se pueden convertir en montones de forma recursiva y luego se puede usar `repararMonton` para filtrar hacia abajo el elemento que está en la raíz hasta ocupar su lugar correcto, combinando así los dos montones más pequeños y la raíz en un montón grande. El caso base es un árbol que consiste en un solo nodo (es decir, una hoja); semejante árbol ya es un montón. El algoritmo que sigue implementa esta idea.

Algoritmo 4.7 Construir un montón

Entradas: Una estructura de montón *H* que no necesariamente tiene la propiedad de árbol en orden parcial.

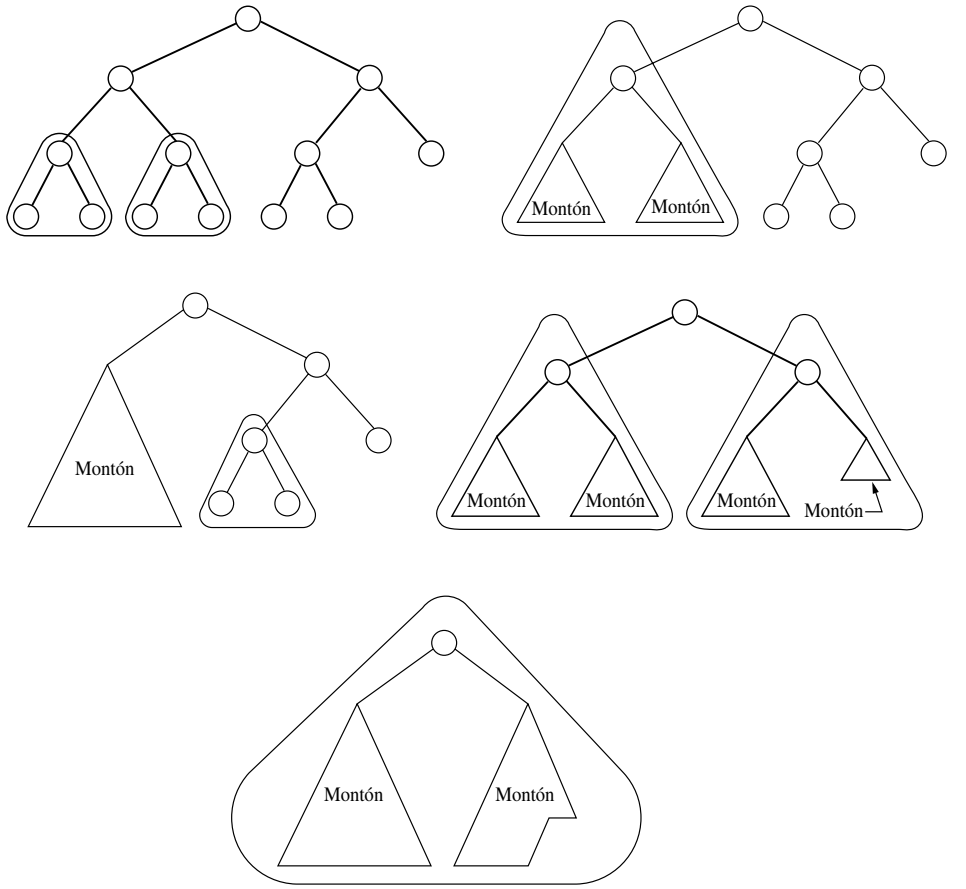


Figura 4.19 Construcción del montón: las hojas son montones. Se invoca el procedimiento `reparar-Monton` para cada subárbol encerrado con una línea.

Salidas: *H* con los mismos nodos reacomodados de modo que satisfagan la propiedad de árbol en orden parcial.

```
void construirMonton(H) // BOSQUEJO
   if (H no es una hoja)
       construirMonton(subárbol izquierdo de H);
       construirMonton(subárbol derecho de H);
       Elemento K = raiz(H);
       repararMonton(H, K);
   return;
```

Si seguimos la pista al trabajo efectuado por este algoritmo de Divide y vencerás, veremos que en realidad comienza a reacomodar elementos primero cerca de las hojas y va subiendo por el árbol. (Es una especie de recorrido en orden posterior.) En la figura 4.19 se presenta una ilustración. El ejercicio 4.38 pide escribir una versión iterativa de `construirMonton`.

Corrección

Teorema 4.13 El procedimiento `construirMonton` establece la propiedad de árbol en orden parcial en su parámetro *H*.

Demostración La demostración es por inducción con estructuras de montón. El caso base es un montón de un solo nodo, que tiene la propiedad de árbol en orden parcial por omisión.

Para montones *H* de más de un nodo, suponemos que el teorema se cumple para los submontones propios de *H*. Las invocaciones recursivas de `construirMonton` se hacen con tales submontones. No hay problema para cumplir con las condiciones previas de las invocaciones recursivas, porque los subárboles de una estructura de montón también son estructuras de montón. Por tanto, por la hipótesis inductiva, *podemos suponer que cumplen con su cometido*; no es necesario ahondar en la recursión.

El último aspecto en materia de corrección es si se satisfacen o no las condiciones previas de la subrutina `repararMonton` en el punto en que se le invoca. Sin embargo, esas condiciones previas no son sino las condiciones posteriores de las dos invocaciones recursivas de `construirMonton`. Por tanto, podemos suponer que `repararMonton` logra su objetivo, que es simplemente el objetivo de la invocación actual de `construirMonton`: hacer que *H* tenga la propiedad de árbol en orden parcial. □

Análisis de peor caso

Una ecuación de recurrencia para `construirMonton` depende del costo de `repararMonton`. Definiendo el tamaño del problema como *n*, el número de nodos de la estructura de montón *H*, vimos que `repararMonton` requiere cerca de $2 \lg(n)$ comparaciones de claves. Denotemos con *r* el número de nodos del submontón derecho de *H*. Entonces tenemos

$$W(n) = W(n - r - 1) + W(r) + 2 \lg(n) \qquad \text{para } n > 1.$$

Aunque los montones están equilibrados en la medida de lo posible, *r* puede bajar hasta $n/3$. Por ello, aunque `construirMonton` es un algoritmo de Divide y vencerás, sus dos subproblemas no son necesariamente iguales. Con la ayuda de matemáticas un tanto arduas es posible resolver la recurrencia para una *n* arbitraria, pero aquí tomaremos un atajo. Primero resolveremos la ecuación para $N = 2^d - 1$, es decir, para el caso de los árboles binarios completos y luego observaremos que, para *n* entre $\frac{1}{2}N$ y *N*, $W(N)$ es una cota superior de $W(n)$.

Con $N = 2^d - 1$, los subárboles izquierdo y derecho tienen el mismo número de nodos, así que la ecuación de recurrencia se convierte en

$$W(N) = 2W(\tfrac{1}{2}(N - 1)) + 2 \lg(N) \qquad \text{para } N > 1.$$

Ahora aplicamos el Teorema maestro (teorema 3.17). Tenemos $b = 2$, $c = 2$ (la diferencia entre $N/2$ y $(N - 1)/2$ carece de importancia), $E = 1$ y $f(N) = 2 \lg(N)$. Si escogemos $\epsilon = 0.1$ (o cualquier fracción menor que 1) demostramos que es válido el caso 1 de ese teorema: $2 \lg(N) \in O(n^{0.9})$. Se sigue que $W(N) \in \Theta(N)$.

Ahora bien, volviendo a *n* general, dado que $N \leq 2n$, $W(n) \leq W(N) \in \Theta(2n) = \Theta(n)$. Así pues, ¡el montón se construye en tiempo lineal! (En el ejercicio 4.39 se presenta un argumento de conteo alterno.)

Todavía no es obvio que Heapsort sea un buen algoritmo; parece requerir espacio extra. Ha llegado el momento de considerar la implementación de un montón.

9	5	7	1	4	3	6

Montón 1

50	24	30	20	21	18	3	12	5	6

Montón 2

Figura 4.20 Almacenamiento de los montones de la figura 4.17

4.8.5 Implementación de un montón y del algoritmo Heapsort

Los árboles binarios por lo regular se implementan como estructuras ligadas en las que cada nodo contiene apuntadores (o algún otro tipo de referencias) a las raíces de sus subárboles. Preparar y usar semejante estructura requiere tiempo y espacio adicionales para los apuntadores. Sin embargo, podemos almacenar y usar un montón de manera eficiente sin apuntadores. En un montón no hay nodos en, digamos, la profundidad d si la profundidad $d - 1$ no está totalmente llena, por lo que un montón se puede almacenar en un arreglo nivel por nivel (comenzando con la raíz), de izquierda a derecha dentro de cada nivel. En la figura 4.20 se muestra la organización de almacenamiento para los montones de la figura 4.17. Para que semejante esquema sea útil, deberemos poder encontrar de forma rápida los hijos de un nodo y también determinar rápidamente si un nodo es una hoja o no. Para usar las fórmulas específicas que vamos a describir es importante que la raíz se almacene con índice 1, no 0, en el arreglo.

Supóngase que se nos da el índice i de un nodo. Entonces podremos usar un argumento de conteo para demostrar que su hijo izquierdo tiene el índice $2i$ y que su hijo derecho tiene el índice $2i + 1$; asimismo, el padre es $\lceil i/2 \rceil$. (La demostración se deja como ejercicio.) Es para simplificar estas fórmulas que usamos índices a partir del 1 al hablar de montones.

La sorprendente característica de Heapsort es que todo el procedimiento de ordenar se puede efectuar en su lugar; los pequeños montones que se crean durante la fase de construcción y, más adelante, el montón y los elementos borrados, pueden ocupar el arreglo E que originalmente contenía el conjunto de elementos en desorden. Durante la fase de eliminación, cuando el montón contiene, digamos, k elementos, ocupará las primeras k posiciones del arreglo. Por ello, sólo necesitamos una variable para marcar el final del arreglo. La figura 4.21 ilustra la forma en que el arreglo se reparte entre el montón y los elementos ordenados. (Cabe señalar que el bosquejo de `repararMonton` del algoritmo 4.6 sólo tenía dos parámetros. La implementación más detallada que sigue tiene cuatro.)

Algoritmo 4.8 Heapsort

Entradas: E, un arreglo no ordenado y $n \geq 1$, el número de elementos. El intervalo de índices es $1, \ldots, n$.

Salidas: E, con sus elementos en orden no decreciente según sus claves.

Comentario: `E[0]` no se usa. Recuerde reservar $n + 1$ posiciones para E.

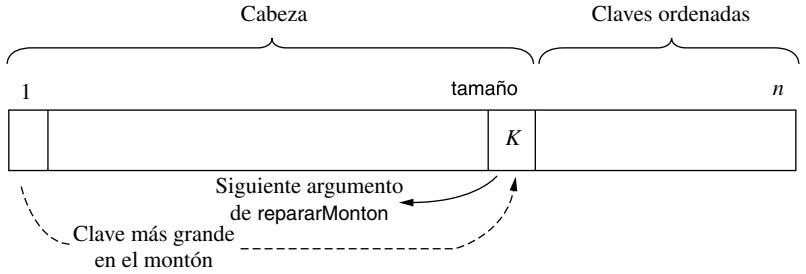


Figura 4.21 El montón y los elementos ordenados en el arreglo

```
void heapSort(Elemento[] E, int n)
   int tamaño;

   construirMonton(E, n);
   // Quitar repetidamente el elemento raíz y reacomodar el montón.
   for(tamaño = n; tamaño ≥ 2; tamaño --;)
      Elemento maxActual = E[1];
      Elemento K = E[tamaño];
      repararMonton(E, tamaño-1, 1, K);
      E[tamaño] = maxActual;
   return;
```

Ahora presentaremos el algoritmo `repararMonton` (algoritmo 4.6) modificado para la implementación con arreglo. Las modificaciones de `construirMonton` (algoritmo 4.7) siguen el mismo patrón, así que las omitiremos.

Algoritmo 4.9 Reparar Montón

Entradas: Un arreglo *E* que representa una estructura de montón; `tamaño`, el número de elementos del montón, `raiz`, la raíz del submontón que se va a reparar (una posición vacante); `K`, el elemento que se insertará en el submontón de manera que restaure la propiedad de árbol en orden parcial. La condición previa es que los submontones cuyas raíces son los hijos izquierdo y derecho de `raiz` tienen la propiedad de árbol en orden parcial.

Salidas: Ya se insertó `K` en el submontón cuya raíz es `raiz` y el submontón tiene la propiedad de árbol en orden parcial.

Procedimiento: Véase la figura 4.22. ■

Análisis de Heapsort

Ahora podemos ver claramente que Heapsort es un ordenamiento en su lugar en términos del espacio de trabajo para los elementos que se van a ordenar. Aunque algunas subrutinas usan recursión, la profundidad de la recursión está limitada a aproximadamente lg *n*, lo cual normalmente no es causa de preocupación. No obstante, podemos recodificar esas subrutinas eliminando la recursión para así trabajar en el lugar (véase el ejercicio 4.38).

```
void repararMonton(Elemento[] E, int tamaño, int raiz, Elemento K)
   int izq = 2 * raiz, der = 2 * raiz + 1;
   if(izq > tamaño)
      E[raiz] = K; // Raiz es una hoja.
   else
      // Determinar cuál es el submontón mayor
      int subMontonMayor;
      if (izq == tamaño)
        subMontonMayor = izq; // No hay submontón derecho.
      else if (E[izq].clave > E[der].clave)
        subMontonMayor = izq;
      else
        subMontonMayor = der;
      // Decidir si filtrar K hacia abajo o no.
      if(K.clave ≥ E[subMontonMayor].clave)
        E[raiz] = K;
      else
        E[raiz] = E[subMontonMayor];
        repararMonton(E, tamaño, subMontonMayor, K);
   return;
```

Figura 4.22 Procedimiento del algoritmo 4.9

Ya vimos en la sección 4.8.4 que el número de comparaciones efectuadas por `construir-Monton` está en $\Theta(n)$. Consideremos ahora el ciclo principal del algoritmo 4.8. Por el lema 4.12, el número de comparaciones que `repararMonton` efectúa con un montón de k nodos es cuando más $2\lfloor \lg k \rfloor$, así que el total para todas las eliminaciones es cuando más $2 \sum_{k=1}^{n-1} \lfloor \lg k \rfloor$. Esta suma se puede acotar con una integral, que adopta la forma de la ecuación (1.15),

$$2 \sum_{k=1}^{n-1} \lfloor \lg k \rfloor \leq 2 \int_1^n (\lg e) \ln x \, dx$$
$$= 2 (\lg e)(n \ln n - n) = 2(n \lg(n) - 1.443\, n).$$

El teorema siguiente sintetiza nuestros resultados.

Teorema 4.14 El número de comparaciones de claves que Heapsort efectúa en el peor caso es $2n \lg n + O(n)$. Heapsort es un algoritmo de ordenamiento $\Theta(n \log n)$.

Demostración La fase de construcción del montón efectúa cuando más $O(n)$ comparaciones y las eliminaciones efectúan cuando más $2n \lg(n)$. □

Heapsort efectúa $\Theta(n \log n)$ comparaciones en promedio y también en el peor caso. (¿Cómo lo sabemos?)

4.8.6 Heapsort acelerado

Recordemos que `repararMonton` maneja el caso en el que la raíz de un montón está vacante pero todos los demás elementos satisfacen la propiedad de árbol en orden parcial. Se debe insertar un elemento nuevo, pero podría no ser lo bastante grande como para ocupar la raíz. El elemento "se filtra hacia abajo" a la izquierda o bien a la derecha, hasta quedar en relación correcta respecto a sus hijos.

Supóngase que una clave está demasiado abajo en el árbol en orden parcial; es decir, que es demasiado grande para la posición que ocupa. Se nos ocurre un procedimiento doble para que ese elemento *suba* como una burbuja por el árbol (es decir, hacia la raíz), haciendo una analogía con una burbuja de aire que sube a través del agua. De hecho, el "burbujeo" es más sencillo, porque el elemento en cuestión sólo tiene un padre; no hay que tomar la decisión de "derecho o izquierdo". El procedimiento `subirMonton` es un complemento natural de `repararMonton` (algoritmo 4.9). Después de precisar los pormenores de `subirMonton`, veremos cómo usar nuestro repertorio ampliado para acelerar Heapsort casi al doble.

Seguimos suponiendo un montón maximizante, porque ése es el tipo que usa Heapsort. En términos más precisos, se proporciona a `subirMonton` un elemento K y una posición "vacante", tal que colocar el elemento en esa posición vacante lo podría dejar demasiado abajo en el montón; es decir, K podría ser mayor que su padre.

El procedimiento permite que los elementos pequeños que estén en el camino de `vacante` a la raíz migren hacia abajo, conforme `vacante` sube, hasta hallar el lugar correcto para el nuevo elemento. (Para un índice i, `padre`$(i) = \lfloor i/2 \rfloor$.) La operación es similar a la acción de Ordenamiento por inserción cuando inserta un elemento nuevo en la porción ordenada del arreglo.

Algoritmo 4.10 Subir (como burbuja) por un montón

Entradas: Un arreglo E que representa una estructura de montón; enteros `raiz` y `vacante`, un elemento K que se insertará en `vacante` o en algún nodo antepasado de `vacante`, hasta `raiz`, de forma tal que se mantenga la propiedad de árbol en orden parcial en E. Como condición previa, E tiene la propiedad de árbol en orden parcial si no se toma en cuenta el nodo `vacante`.

Salidas: Se ha insertado `K` en el submontón cuya raíz es `raiz` y el submontón tiene la propiedad de árbol en orden parcial.

Comentario: La estructura de E no se altera, pero sí cambia el contenido de sus nodos.

```
void subirMonton(Elemento[] E, int raiz, Elemento K, int vacante)
   if(vacante == raiz)
      E[vacante] = K;
   else
      int padre = vacante / 2;
      if(K.clave ≤ E[padre].clave)
        E[vacante] = K;
      else
        E[vacante] = E[padre];
        subirMonton(E, raiz, K, padre);
```

El "burbujeo" de un elemento por el montón empleando `subirMonton` sólo requiere una comparación por nivel subido. Se puede usar el algoritmo 4.10 para apoyar la inserción en un montón (véase el ejercicio 4.41).

Si combinamos `subirMonton` con un `repararMonton` ligeramente modificado podremos reducir el número de comparaciones que Heapsort efectúa en un factor de aproximadamente 2. Esto hace a Heapsort muy competitivo con Mergesort en términos del número de comparaciones; sus primeros términos de peor caso ahora tienen el mismo coeficiente (uno). El número de traslados de elementos no se reduce más, pero esa medida ya era comparable con la de Mergesort. Heapsort tiene la ventaja de que no requiere un arreglo auxiliar, como Mergesort.

La idea principal es simple. El filtrado de un elemento hacia abajo con `repararMonton` requiere dos comparaciones por nivel. Sin embargo, podemos evitar una de esas comparaciones la que se hace con *K*, el elemento que se está filtrando sin perturbar la propiedad de árbol en orden parcial. Es decir, comparamos los hijos izquierdo y derecho de `vacante` y subimos el elemento mayor a `vacante` (suponiendo un montón maximizante). Ahora `vacante` baja a la posición que ocupaba el hijo que subió. Esto sólo nos cuesta una comparación por nivel, en lugar de dos. Llamaremos por ahora a esta variante `repararMonton` *arriesgado*.

Al no comparar el hijo mayor con *K*, nos arriesgamos a bajar demasiado en el árbol y "promover" elementos menores que *K*. Pero si llega a suceder esto, podemos usar `subirMonton` para *subir K* a su posición correcta, con un costo de una comparación por nivel. Puesto que el "burbujeo" de *K* árbol arriba no puede requerir más comparaciones que las que ahorramos al bajar, no hay forma de perder. Sea h la altura del montón de n nodos. El `repararMonton` normal cuesta $2h$ comparaciones en el peor caso. Ejecutar el `repararMonton` arriesgado bajando por todo el árbol sólo cuesta h comparaciones. Ahora `subirMonton` requerirá cuando más h comparaciones y podría requerir muchas menos.

Recordemos cómo funciona `borrarMax`. Después de quitar el elemento que está en la raíz, reinsertamos un elemento *K* tomado del fondo del montón. Por ello, es probable que *K* sea un nodo más bien pequeño, que seguramente no subirá mucho por el árbol al burbujear. De hecho, en promedio, este método modificado ahorra la mitad de las comparaciones que requiere `borrarMax`. Sin embargo, en el peor caso, *K* subirá casi hasta la raíz del montón y se perderá casi todo lo que se había ahorrado. ¿Hay alguna forma de hacer menos comparaciones incluso en el peor caso? Invitamos al lector a meditar acerca de este problema antes de continuar.

■ ■ ■

La solución es una sorprendente aplicación de Divide y vencerás. Demos a "`repararMonton` arriesgado" un nombre más descriptivo: `promover`. Usaremos `promover` para filtrar la posición `vacante` la mitad del camino hacia la base del árbol; es decir, $\frac{h}{2}$ niveles. En la figura 4.23 se presenta una ilustración. Ahora probamos si *K* es mayor que el padre de `vacante`. Si lo es, iniciaremos `subirMonton` desde este nivel y el costo será de cuando más otras $\frac{h}{2}$ comparaciones, para un total de h. Si no, buscaremos recursivamente la posición correcta de *K* en el submontón cuya raíz es `vacante`. Este submontón tiene una altura de sólo $\frac{h}{2}$. Es decir, ejecutamos `promover` para filtrar la posición `vacante` otros $h/4$ niveles y comparamos *K* con el padre de `vacante`. Si *K* es mayor, entonces iniciamos `subirMonton` desde este nivel (profundidad $3\frac{h}{4}$). Sin embargo, ahora `subirMonton` puede hacer subir *K cuando más* hasta la profundidad $\frac{h}{2}$, porque ya vimos que *K* era menor (o igual) que el padre del nodo que está a la profundidad $\frac{h}{2}$. Si *K* sigue siendo menor, se ejecutará `promover` para bajar otros $h/8$ niveles, y así sucesivamente.

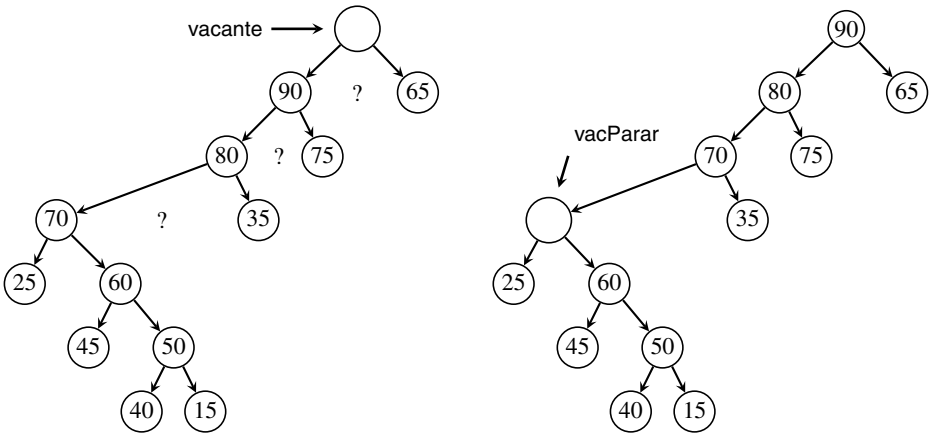


Figura 4.23 La invocación de `promover` con el montón de la izquierda (no se muestran los nodos no cercanos al camino) y con $h = 6$ y `altParar` $= 3$ produce el montón de la derecha.

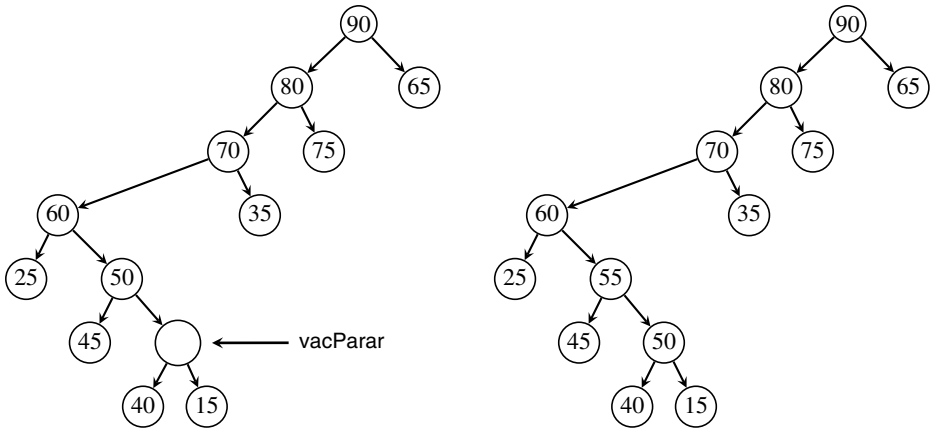


Figura 4.24 Continuación de la figura 4.23. Se debe reinsertar $K = 55$. Puesto que $K < 70$, se invoca `promover` con $h = 3$ y `altParar` $= 1$ para producir el montón de la izquierda. Luego se invoca `subirMonton` para dar la configuración final de la derecha.

Ejemplo 4.2 `repararMonton` acelerado en acción

Supóngase que el elemento a reinsertar en el montón es $K = 55$ y que partimos del montón que se muestra parcialmente en la mitad izquierda de la figura 4.23. Como ya se dijo, `promover` efectúa tres comparaciones y promueve 90, 80 y 70 sin jamás inspeccionar K, lo que lleva a la situación de la derecha.

Ahora se compara K con 70, el padre del nodo `vacParar` actual y se determina que es más pequeño. Por tanto, se invoca `promover` otra vez para bajar el nodo vacante de la altura 3 a la altura 1; es decir, $60 > 25$, así que se promueve 60, luego $50 > 45$, así que se promueve 50. Esto lleva a la situación de la izquierda en la figura 4.24.

Ahora se compara K (55) con 50 y se ve que es mayor, por lo que deberá subir a una posición más alta en el árbol, pero no más allá de la altura 3, en vista de la comparación anterior con 70. La operación de `repararMonton` concluye con el uso de `subirMonton` para que K (55) retroceda hacia arriba a lo largo del camino que se siguió al bajar y produzca el montón final que se muestra a la derecha en la figura 4.24. ■

Algoritmo 4.11 Reparar Montón acelerado

Entradas: E, un arreglo en el que se almacena una estructura de montón, donde `E[1]` es la raíz de todo el montón; n, el número de elementos de E; K, un elemento que se debe ubicar correctamente en el montón; `vacante`, un índice de E que indica una posición donde podría ir K y que actualmente no contiene ningún elemento; h, la altura máxima posible del submontón cuya raíz es `vacante` (dado que el último nivel podría estar incompleto, la altura podría variar en 1).

La condición previa es que se satisfaga la propiedad de árbol en orden parcial en los submontones propios del submontón cuya raíz es `vacante`.

Salidas: E con K insertado en el submontón cuya raíz es `vacante`, de modo que se cumpla la propiedad de árbol en orden parcial para dicho submontón.

Comentario: El procedimiento `borrarMax` invocaría a `repararMontonRapido` con los parámetros `vacante` $= 1$ y $h = \lceil \lg(n + 1) \rceil - 1$.

Procedimiento: Véase la figura 4.25. ■

Análisis

En esencia, se efectúa una comparación cada vez que `vacante` sube o baja un nivel debido a la acción de `subirMonton` o bien de `promover`. El primer `promover` hace que `vacante` baje $\frac{h}{2}$ niveles. Si en ese momento se invoca `subirMonton`, hará que `vacante` suba cuando más $\frac{h}{2}$ niveles y con ello terminará la labor de `repararMontonRapido`. Si no se invocó `subirMonton`, fue porque se determinó que K era menor que el elemento que está en el padre de `vacante`, y se invocará de nuevo `promover` para bajar `vacante` otros $h/4$ niveles. Si en este momento se invoca `subirMonton`, éste hará que `vacante` suba cuando más $h/4$ niveles, porque ya vimos que K era menor (o igual) que el padre del nodo que está a la profundidad $\frac{h}{2}$. Por tanto, el costo total sigue estando limitado a aproximadamente h en este caso. Este patrón continúa, bajando otros $h/8$ niveles, luego $h/16$ niveles y así sucesivamente. Entonces, el número total de comparaciones efectuadas por todas las invocaciones de `promover` y posiblemente una invocación de `subirMonton` será de $h + 1$ (contemplando redondeo si h es impar).

Supóngase que nunca se invoca `subirMonton`, así que `repararMontonRapido` llega a su caso base (y posiblemente necesitará otras dos comparaciones en el caso base). Entonces, en última instancia, `repararMontonRapido` habrá efectuado $\lg(h)$ verificaciones para determinar si debe cambiar de dirección o no. Si sumamos éstas a las comparaciones efectuadas por `promover` y `subirMonton`, tendremos aproximadamente $h + \lg(h)$ comparaciones en total.

Procediendo de manera más formal, la ecuación de recurrencia es

$$T(h) = \lceil \tfrac{1}{2} h \rceil + \text{máx}\,(\lceil \tfrac{1}{2} h \rceil, 1 + T(\lfloor \tfrac{1}{2} h \rfloor)) \qquad T(1) = 2.$$

```
void repararMontonRapido(Elemento[] E, int n, Elemento K, int
     vacante, int h)
   if (h ≤ 1)
      Procesar montón de altura 0 o 1.
   else
      int altParar = h/2;
      int vacParar = promover(E, altParar, vacante, h);
      // vacParar es la nueva posición vacante, a la altura altParar.
      int padreVac = vacParar / 2;
      if (E[padreVac].clave ≤ K.clave)
        E[vacParar] = E[padreVac];
        subirMonton(E, vacante, K, padreVac);
      else
        repararMontonRapido(E, n, K, vacParar, altParar);

int promover(Elemento[] E, int altParar, int vacante, int h)
   int vacParar;
   if (h ≤ altParar)
      vacParar = vacante;
   else if (E[2*vacante].clave ≤ E[2*vacante+1].clave)
      E[vacante] = E[2*vacante+1];
      vacParar = promover(E, altParar, 2*vacante+1, h−1);
   else
      E[vacante] = E[2*vacante];
      vacParar = promover(E, altParar, 2*vacante, h−1);
   return vacParar;
```

Figura 4.25 Procedimiento para el algoritmo 4.11

Si suponemos que $T(h) \geq h$, como sucede en el caso base, la recurrencia se simplifica a

$$T(h) = \lceil \tfrac{1}{2}h \rceil + 1 + T(\lfloor \tfrac{1}{2}h \rfloor) \qquad T(1) = 2.$$

Podemos obtener la solución a partir del árbol de recursión (véase la sección 3.7). También podríamos conjeturar la solución calculando unos cuantos casos pequeños, y verificándola después por inducción. (En el ejercicio 4.44 se presenta la identidad clave.)

$$T(h) = h + \lceil \lg(h+1) \rceil.$$

Así pues, podemos ejecutar `borrarMax` con un montón de n elementos efectuando $\lg(n+1) + \lg\lg(n+1)$ comparaciones aproximadamente, en lugar de $2\lg(n+1)$. El teorema siguiente resume el resultado.

Teorema 4.15 El número de comparaciones que efectúa Heapsort acelerado empleando la subrutina `repararMontonRapido` es $n\lg(n) + \Theta(n \log\log(n))$, en el peor caso. □

Algoritmo	Peor caso	Promedio	Consumo de espacio
Ordenamiento por inserción	$n^2/2$	$\Theta(n^2)$	En su lugar
Quicksort	$n^2/2$	$\Theta(n \log n)$	Espacio extra proporcional a $\log n$
Mergesort	$n \lg n$	$\Theta(n \log n)$	Espacio extra proporcional a n para fusionar
Heapsort	$2n \lg n$	$\Theta(n \log n)$	En su lugar
Aeapsort acel.	$n \lg n$	$\Theta(n \log n)$	En su lugar

Tabla 4.1 Resultados del análisis de cuatro algoritmos de ordenamiento. Los datos son números de comparaciones e incluyen únicamente el primer término.

4.9 Comparación de cuatro algoritmos para ordenar

En la tabla 4.1 se sintetizan los resultados del análisis de comportamiento de los cuatro algoritmos de ordenamiento que hemos visto hasta ahora. Aunque Mergesort se acerca a la optimidad en el peor caso, hay algoritmos que efectúan menos comparaciones. La cota inferior obtenida en la sección 4.7 es muy buena. Sabemos que es exacta para algunos valores de n; es decir, bastan $\lceil \lg n! \rceil$ comparaciones para ordenar, con ciertos valores de n. También sabemos que no bastan $\lceil \lg n! \rceil$ comparaciones para todos los valores de n. Por ejemplo, $\lceil \lg 12! \rceil = 29$, pero se ha demostrado que son necesarias (y suficientes) 30 comparaciones para ordenar 12 elementos en el peor caso. En las Notas y referencias al final del capítulo se mencionan obras que tratan algoritmos de ordenamiento cuyo comportamiento de peor caso es cercano a la cota inferior.

4.10 Shellsort

La técnica empleada por Shellsort (así llamado por su inventor, Donald Shell) es interesante, el algoritmo es fácil de programar y se ejecuta con rapidez aceptable. Su análisis, en cambio, es muy difícil e incompleto.

4.10.1 El algoritmo

Shellsort ordena un arreglo E de n elementos ordenando sucesivamente subsucesiones cuyos elementos están entremezclados en todo el arreglo. Las subsucesiones a ordenar están determinadas por una sucesión, $h_t, h_{t-1}, \ldots, h_1$, de parámetros llamados *incrementos*. Supóngase, por ejemplo, que el primer incremento, h_t, es 6. Entonces el arreglo se divide en seis subsucesiones, como sigue:

1. `E[0], E[6], E[12],...`
2. `E[1], E[7], E[13],...`
3. `E[2], E[8], E[14],...`
4. `E[3], E[9], E[15],...`
5. `E[4], E[10], E[16],...`
6. `E[5], E[11], E[17],...`

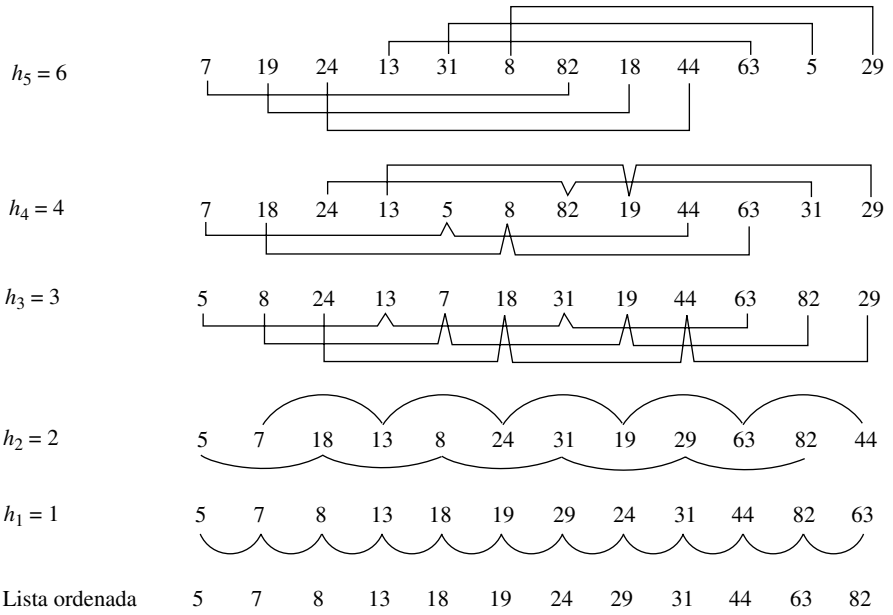


Figura 4.26 Shellsort: observe que sólo se intercambian dos pares de elementos en la última pasada

Como vemos, las subsucesiones se forman saltando de 6 en 6 por el arreglo, en este ejemplo, y de h_t en h_t, en general.

Una vez ordenadas estas subsucesiones, se usa el siguiente incremento, h_{t-1}, para volver a dividir el arreglo en subsucesiones, ahora con elementos tomados de h_{t-1} y se vuelven a ordenar las subsucesiones. El proceso se repite para cada incremento. El incremento final, h_1, siempre es 1, así que al final todo el arreglo estará ordenado. La figura 4.26 ilustra la acción de este método con un arreglo pequeño.

La descripción informal de Shellsort deberá suscitar varias preguntas. ¿Qué algoritmo debe usarse para ordenar las subsucesiones? Considerando que el último incremento es 1 y que en la última pasada se ordena todo el arreglo, ¿es Shellsort más eficiente que el algoritmo empleado para ordenar las subsucesiones?, ¿se puede escribir el algoritmo de modo que se reduzca al mínimo toda la "contabilidad" que al parecer es necesaria para controlar el ordenamiento de todas las subsucesiones?, ¿qué incrementos debemos usar?

Abordaremos primero las dos primeras preguntas. Como muestra el ejemplo de la figura 4.26, si las últimas pasadas se efectúan empleando incrementos pequeños, pocos elementos estarán en desorden gracias al trabajo efectuado en pasadas anteriores. Por tanto, Shellsort podría ser eficiente si, y de hecho sólo sería eficiente si, el método empleado para ordenar subsucesiones es uno que efectúa muy poco trabajo si el arreglo ya está ordenado o casi ordenado. El Ordenamiento por inserción (sección 4.2) tiene esta propiedad, pues efectúa sólo $n - 1$ comparaciones si el arreglo está totalmente ordenado, es sencillo de programar y realiza poco procesamiento fijo.

Supóngase ahora que Shellsort está usando un incremento h y debe ordenar h subsucesiones, cada una de las cuales contiene aproximadamente n/h elementos. Si queremos que cada subsucesión quede totalmente ordenada antes de que se comience a trabajar con la siguiente, el algoritmo necesitaría saber cuáles subsucesiones ya se ordenaron y cuáles deben ordenarse aún. Evitamos esta "contabilidad" haciendo que el algoritmo efectúe una pasada por todo el arreglo (para cada incremento), entremezclando el trabajo que efectúa con todas las subsucesiones. Los elementos consecutivos de una subsucesión están separados h celdas en vez de una sola celda. Recuerde el Ordenamiento por inserción (algoritmo 4.1) y su subrutina interactiva `desplaVac`. Generalmente, "1" en `desplaVac` se sustituye por "h" en `desplaVacH`, la subrutina que usa Shellsort.

Algoritmo 4.12 Shellsort

Entradas: E, un arreglo no ordenado de elementos; $n \geq 0$, el número de elementos; una sucesión de incrementos decrecientes, $h_t, h_{t-1}, \ldots, h_1$, donde $h_1 = 1$; y t *como* el número de incrementos. El intervalo de índices del arreglo E es $0, \ldots, n - 1$.

Salidas: E, con sus elementos en orden no descendente según sus claves.

Comentario: La sucesión de incrementos podría calcularse, en lugar de proporcionarse como entrada.

```
void shellSort(Elemento[] E, int n, int[] h, int t)
   int indicex, s;

   for (s = t; s ≥ 1; s --)
      for (indicex = h[s]; indicex < n; indicex ++)
      // indicex inicia en el segundo elemento de la subsucesión 0.
         Elemento actual = E[indicex];
         Clave x = actual.clave;
         int posX = desplaVacH(E, h[s], indicex, x);
         E[posX] = actual;
   return;

// Version de desplaVac para Shellsort, usa incremento h.
int desplaVacH(Elemento[] E, int h, int indicex, Clave x)
   int vacante, posX;
   vacante = indicex;
   posX = 0; // Suponemos fracaso.
   while (vacante ≥ h)
      // vacante-h es el índice anterior dentro de la subsucesión
         actual.
      if (E[vacante-h].clave ≤ x)
         posX = vacante; // Éxito.
         break;
      E[vacante] = E[vacante-h];
      vacante −= h; // Seguir buscando.
   return posX;
```

Recordemos que Ordenamiento por inserción era lento porque `desplaVac` eliminaba cuando más una inversión después de cada comparación. Aunque después de cada comparación en Shellsort `desplaVacH` elimina cuando más una inversión de la subsucesión que está ordenando, tiene la oportunidad de eliminar hasta h inversiones de todo el arreglo con cada comparación, porque hace que se trasladen elementos a una posición distante. Por ello, existe la posibilidad de que el comportamiento promedio de Shellsort esté en $o(n^2)$. La eficiencia de Shellsort se debe al hecho de que ordenar con un incremento, digamos k, no revierte en absoluto el trabajo efectuado previamente cuando se usó un incremento distinto, digamos h. En términos más precisos, decimos que una lista está ordenada por h si `E[i]`$\leq$ `E[i + h]` para $0 \leq i < n - h$; en otras palabras, si todas las subsucesiones formadas por cada h-ésimo elemento están ordenadas. Ordenar por h un arreglo implica ordenar subsucesiones empleando como incremento h.

Teorema 4.16 Si un arreglo ordenado por h se ordena por k, seguirá estando ordenado por h.

Demostración Véanse las Notas y referencias al final del capítulo. Vale la pena examinar la figura 4.26 para ver que el teorema se cumple para el ejemplo ilustrado. □

4.10.2 Análisis y comentarios

El número de comparaciones efectuadas por Shellsort es función de la sucesión de incrementos empleada. Un análisis completo es en extremo difícil y requiere respuestas a algunos problemas matemáticos que todavía no se han resuelto. Por tanto, no se ha determinado aún la sucesión de incrementos óptima, pero sí se han estudiado exhaustivamente algunos casos específicos. Uno de ellos es el caso en que $t = 2$, es decir, en el que se usan exactamente dos incrementos, h y 1. Se ha demostrado que el mejor valor para h es aproximadamente $1.72\sqrt[3]{n}$, y que con este valor el tiempo de ejecución medio es proporcional a $n^{5/3}$. Esto podría parecer sorprendente, ya que usar el incremento 1 equivale a ejecutar el Ordenamiento por inserción, que tiene un comportamiento promedio $\Theta(n^2)$; basta efectuar una pasada preliminar por el arreglo con incremento h para reducir el orden asintótico del tiempo de ejecución. Si usamos más de dos incrementos, podremos mejorar aún más el tiempo de ejecución.

Se sabe que si los incrementos son $h_k = 2^k - 1$, para $1 \leq k \leq \lfloor \lg n \rfloor$, el número de comparaciones efectuadas en el peor caso está en $O(n^{3/2})$. Estudios empíricos (con valores de n tan altos como 250,000) han demostrado que otro juego de incrementos da pie a programas que se ejecutan con gran rapidez. Éstos están definidos para $h_i = (3^i - 1)/2$ para $1 \leq i \leq t$, donde t es el entero más pequeño tal que $h_{t+2} \geq n$. Es fácil calcular iterativamente estos incrementos. Podemos obtener h_t al principio del ordenamiento utilizando la relación $h_{s+1} = 3h_s + 1$ y comparando los resultados con n. En lugar de almacenar todos los incrementos, podemos recalcularlos en orden inverso durante el ordenamiento empleando la fórmula $h_s = (h_{s+1} - 1)/3$.

Se ha demostrado que, si los incrementos consisten en todos los enteros de la forma 2^i3^j menores que n (empleados en orden decreciente), el número de comparaciones efectuadas está en $O(n(\log n)^2)$. Se sabe o se espera que los tiempos de ejecución de peor caso con los otros juegos de incrementos tengan orden asintótico más alto. No obstante, debido al gran número de enteros de la forma 2^i3^j, se efectuarán más pasadas por el arreglo, y por ende el procesamiento fijo será mayor, con estos incrementos que con otros. Por ello, no resultan muy útiles a menos que n sea relativamente grande.

Es evidente que Shellsort es un ordenamiento en su lugar. Aunque el análisis del algoritmo dista mucho de ser completo, no se sabe cuáles incrementos son los mejores, su rapidez y sencillez hacen que sea una buena opción en la práctica.

4.11 Ordenamiento por base

Para los algoritmos de ordenamiento de las secciones 4.2 a 4.10, sólo se hizo un supuesto acerca de las claves: son elementos de un conjunto linealmente ordenado. La operación básica de los algoritmos es una comparación de dos claves. Si hacemos más supuestos acerca de las claves, podremos considerar algoritmos que realizan otras operaciones con ellos. En esta sección estudiaremos unos cuantos de esos algoritmos, llamados "ordenamientos de cubetas", "ordenamientos por base" y "ordenamientos por distribución".

4.11.1 Cómo usar las propiedades de las claves

Supóngase que las claves son nombres y están impresos en tarjetas, con un nombre por tarjeta. Para colocar en orden alfabético las tarjetas manualmente, podríamos dividirlas primero en 26 montones según la primera letra del nombre, o en menos montones con varias letras en cada uno; ordenar alfabéticamente las tarjetas de cada montón empleando algún otro método, tal vez similar a Ordenamiento por inserción; y por último combinar los montones ordenados. Si todas las claves son enteros decimales de cinco dígitos, podríamos dividirlos en 10 pilas según el primer dígito. Si son enteros entre 1 y m, para alguna m, podríamos hacer un montón para cada uno de los k intervalos $[1, m/k]$, $[m/k + 1, 2m/k]$, etc. En cada uno de estos ejemplos, las claves se reparten entre diferentes montones después de examinar letras o dígitos individuales de una clave o después de comparar claves con valores predeterminados. Luego los montones se ordenan individualmente y se recombinan. Los algoritmos que ordenan empleando tales métodos no pertenecen a la clase que consideramos antes porque para usarlos necesitamos saber algo acerca de la estructura o del intervalo de las claves.

Presentaremos un algoritmo detallado de *ordenamiento por base* más adelante. Para distinguir el algoritmo específico de otros del mismo tipo, usaremos el término "ordenamientos de cubetas" para la clase general de algoritmos.

¿Qué tan rápidos son los ordenamientos de cubetas?

Un ordenamiento de cubetas tiene tres fases:

1. distribuir claves,
2. ordenar cubetas individualmente,
3. combinar cubetas.

El tipo de trabajo que se efectúa en cada fase es diferente, por lo que aquí no va a funcionar bien nuestro enfoque usual de escoger una operación básica y contarla. Supóngase que hay k cubetas.

Durante la fase de distribución, el algoritmo examina cada clave una vez (sea examinando un campo de bits en particular o comparando la clave con algún número constante de valores preestablecidos). Luego se efectúa cierto trabajo para indicar en qué cubeta va la clave. Esto podría implicar copiar el elemento o establecer algunos índices o apuntadores. El número de operaciones efectuadas por una implementación razonable de la primera fase deberá estar en $\Theta(n)$.

Para ordenar las cubetas, supóngase que usamos un algoritmo que ordena por comparación de claves efectuando, digamos, $S(m)$ comparaciones con una cubeta que contiene m elementos. Sea n_i el número de elementos de la i-ésima cubeta. El algoritmo efectúa $\sum_{i=1}^{k} S(n_i)$ comparaciones durante la segunda fase.

La tercera fase, combinar las cubetas, podría requerir, en el peor de los casos, copiar todos los elementos de las cubetas en una lista; la cantidad de trabajo efectuada está en $O(n)$.

Así pues, la mayor parte del trabajo se efectúa al ordenar cubetas. Supóngase que $S(m)$ está en $\Theta(m \log m)$. Entonces, si las claves están distribuidas de manera uniforme entre las cubetas, el algoritmo efectuará aproximadamente $ck(n/k) \lg(n/k) = cn \lg(n/k)$ comparaciones de claves en la segunda fase, donde c es una constante que depende del algoritmo de ordenamiento empleado en las cubetas. Si aumentamos k, el número de cubetas, reduciremos el número de comparaciones efectuadas. Si escogemos $k = n/10$, entonces se efectuarán $n \lg 10$ comparaciones y el tiempo de ejecución del ordenamiento de cubetas será lineal en n, suponiendo que las claves estén distribuidas de manera uniforme y que el tiempo de ejecución de la primera fase no dependa de k. (Hay que señalar, como advertencia, que cuantos menos elementos haya por cubeta, menos probable será que la distribución sea uniforme.) Sin embargo, en el peor caso, todos los elementos quedarán en una sola cubeta y en la segunda fase se ordenará toda la lista, convirtiendo todo el trabajo de la primera y la última fases en procesamiento fijo desperdiciado. Así pues, en el peor caso, un ordenamiento de cubetas sería muy ineficiente. Si se conoce con antelación la distribución de las claves, se podrá ajustar el intervalo de claves que van en cada cubeta de modo que todas las cubetas reciban un número aproximadamente igual de elementos.

La cantidad de espacio que necesita un ordenamiento de cubetas depende de cómo se almacenan las cubetas. Si cada cubeta consiste en un conjunto de posiciones sucesivas (por ejemplo, un arreglo), se deberá asignar suficiente espacio a cada una como para contener el número máximo de elementos que podrían ir en una cubeta, o sea n. Por tanto, se usarían kn posiciones para ordenar n elementos. A medida que aumenta el número de cubetas, aumenta la velocidad del algoritmo pero también aumenta la cantidad de espacio utilizada. Sería mejor usar listas ligadas; sólo se usaría $\Theta(n + k)$ espacio (para n elementos más ligas y una cabeza de lista para cada cubeta). Para repartir las claves entre las cubetas sería necesario construir nodos de lista. Pero entonces, ¿cómo se ordenarían los elementos de cada cubeta? Es fácil implementar Quicksort y Mergesort, dos de los algoritmos más rápidos que hemos visto, para ordenar listas ligadas (véanse los ejercicios 4.22 y 4.28). Si el número de cubetas es grande, el número de elementos en cada una generalmente será pequeño y se podría usar un algoritmo más lento. También es fácil modificar Ordenamiento por inserción para ordenar los elementos de una lista ligada (véase el ejercicio 4.11). Con aproximadamente n/k elementos por cubeta, Mergesort efectuará cerca de $(n/k)(\lg(n) - \lg(k))$ comparaciones en promedio con cada cubeta, o sea, $n(\lg(n) - \lg(k))$ comparaciones en total. Aquí también, a medida que aumenta k, aumenta la velocidad pero también aumenta el espacio ocupado.

El lector podría preguntarse por qué no usamos recursivamente un algoritmo de ordenamiento de cubetas para crear cubetas cada vez más pequeñas. Hay varias razones. La contabilidad pronto se volvería excesiva; sería preciso apilar y desapilar a menudo apuntadores que indican dónde comienzan las diversas cubetas, así como la información necesaria para recombinar los elementos en una sola lista. Debido a la contabilidad necesaria para efectuar cada invocación recursiva, el algoritmo no debe confiar en que al final se tendrá un solo elemento por cubeta, utilizando en última instancia otro algoritmo para ordenar cubetas pequeñas. Por ello, si se usa desde un principio un número relativamente grande de cubetas, no habría mucho que ganar y sí mucho que per-

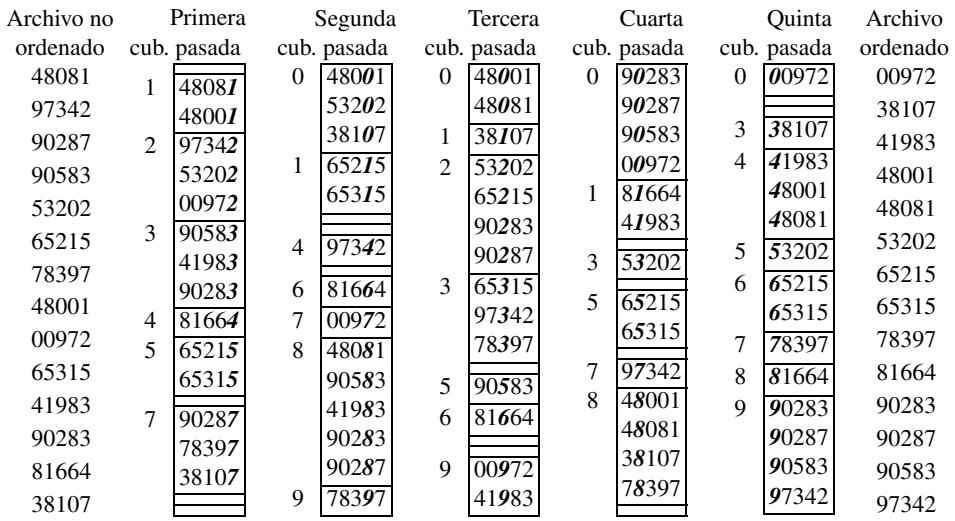


Figura 4.27 Ordenamiento por base

der si se ordenan las cubetas recursivamente. Por otra parte, aunque repartir recursivamente las claves entre las cubetas no resulta eficiente, podemos salvar algo muy útil de esta idea.

4.11.2 Ordenamiento por base

Supóngase que las claves son números de cinco dígitos. Un algoritmo recursivo, como acaba de sugerirse, podría repartir primero las claves entre 10 cubetas según el dígito de la extrema izquierda (el más significativo), y luego repartir las claves de cada cubeta entre otras 10 cubetas según el siguiente dígito más significativo, y así sucesivamente. Las cubetas no se combinarían sino hasta que estuvieran totalmente ordenadas, de ahí la gran cantidad de trámites contables. Es curioso que si las claves se reparten primero entre cubetas según los dígitos (o bits, o letras, o campos) *menos significativos*, las cubetas podrán combinarse en orden antes de efectuar la distribución según el siguiente dígito. Se ha eliminado totalmente el problema de ordenar las cubetas. Si hay, digamos, cinco dígitos en cada clave, entonces el algoritmo repartirá las claves entre cubetas y combinará las cubetas cinco veces. Las claves se distribuirán según cada posición de dígito por turno, de derecha a izquierda, como se ilustra en la figura 4.27.

¿Esto siempre funciona?, en la pasada final, cuando se colocan dos claves en la misma cubeta porque las dos comienzan con, digamos, 9, ¿qué nos garantiza que estarán en el orden correcto una en relación con la otra?, en la figura 4.27, las claves 90283 y 90583 difieren únicamente en el tercer dígito y se colocan en la misma cubeta en todas las pasadas con excepción de la tercera. Después de la tercera pasada, en tanto las cubetas se combinen en orden y no se altere el orden relativo de dos claves que se colocan en la misma cubeta, dichas claves se mantendrán en el orden correcto una respecto a la otra. En general, si la posición de dígito más a la izquierda en la que dos claves difieren es la *i*-ésima posición (a partir de la derecha), quedarán en el orden correc-

to una respecto a la otra después de la *i*-ésima pasada. Esta afirmación se puede demostrar por inducción directa.

Este método de ordenamiento es el que usan las máquinas que ordenan tarjetas. En las máquinas viejas, la máquina se encargaba del paso de distribución; el operador recogía los montones después de cada pasada y los juntaba en uno solo para la siguiente pasada.

La distribución en montones, o cubetas, puede controlarse con una columna de una tarjeta, una posición de dígito o un campo de bit de la clave. El algoritmo se llama *Ordenamiento por Base* porque trata las claves como números expresados en una base dada. En el ejemplo de la figura 4.27, la base es 10. Si las claves son enteros positivos de 32 bits, el algoritmo podría usar, digamos, campos de cuatro bits, tratando implícitamente las claves como números base 16. El algoritmo las repartiría entre 16 cubetas. Así pues, la base es también el número de cubetas. En el algoritmo de Ordenamiento por Base que sigue, suponemos que la distribución se efectúa según campos de bits. Los campos se extraen de las claves comenzando por los bits de orden bajo. Si es posible, el número de campos se mantiene constante y no depende de n, el número de elementos de la entrada. En general, esto requiere que la base (número de cubetas) aumente al incrementar n. Una opción versátil es el valor $2^w \leq n$ más grande, donde w es un entero. Así, cada campo tiene w bits de anchura. Si las claves están distribuidas de forma densa (es decir, dentro de un intervalo proporcional a algún polinomio en n), esta estrategia produce un número constante de campos.

La estructura de datos se ilustra en la figura 4.28 para la tercera pasada del ejemplo de la figura 4.27. Observe que cada lista `cubetas` está en orden inverso porque los elementos nuevos se anexan al principio de la lista anterior. Sin embargo, el procedimiento `combinar` los invierte una vez más al combinarlos, de modo que la lista combinada final está en el orden correcto.

Algoritmo 4.13 Ordenamiento por Base

Entradas: `L`, una lista no ordenada; `base`, el número de cubetas para el reparto, y `numCampos`, el número de campos de la clave según la cual se efectúa la distribución.

Salidas: La lista ordenada, `nuevaL`.

Comentario: El procedimiento `distribuir` invierte las listas que entran en las cubetas y `combinar` las vuelve a invertir al sacarlas (y tiene su ciclo en el orden opuesto), de modo que la combinación conserva el orden deseado. Se usan operaciones del TDA Lista para manipular listas ligadas (véase la figura 2.3).

```
Lista ordenBase(Lista L, int base, int numCampos)
   Lista[] cubetas = new Lista[base];
   int campo; // número de campo dentro de la clave.
   Lista nuevaL;

   nuevaL = L;
   for (campo = 0; campo < numCampos; campo ++)
       Inicializar el arreglo cubetas con listas vacías.
       distribuir(nuevaL, cubetas, base, campo);
       nuevaL = combinar(cubetas, base);
   return nuevaL;
```

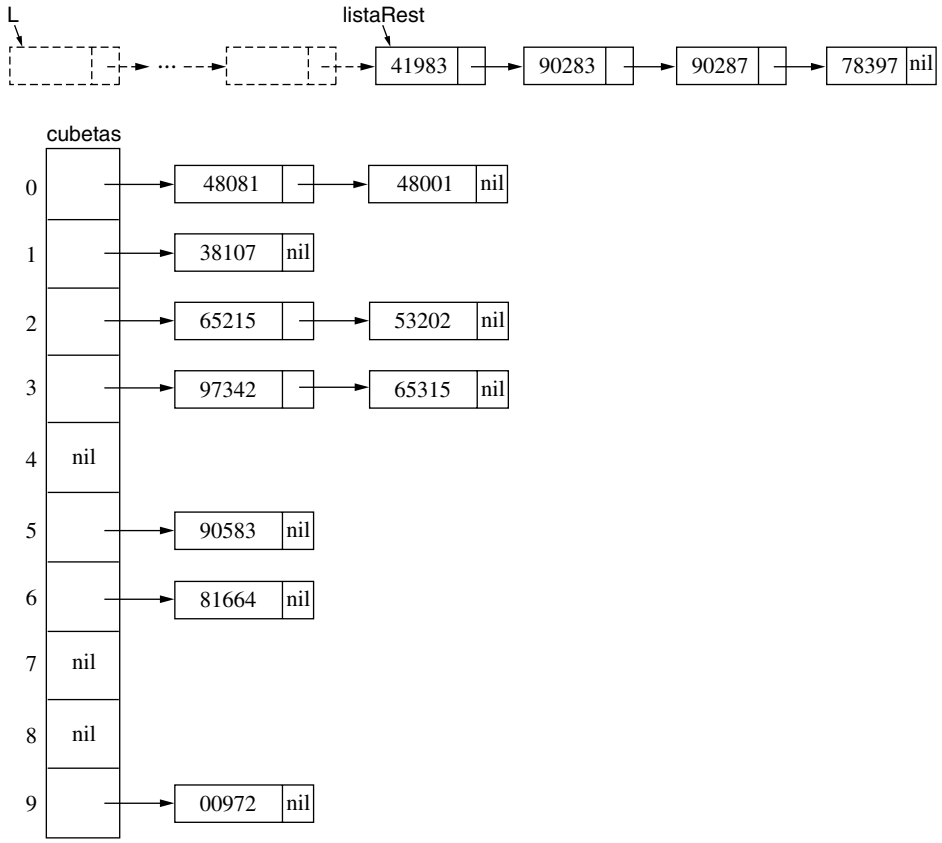


Figura 4.28 La estructura de datos para Ordenamiento por Base durante la distribución según el tercer dígito

```
void distribuir(Lista L, Lista[] cubetas, int base, int campo)
   // Repartir claves entre cubetas.
   Lista listaRest;

   listaRest = L;
   while (listaRest ≠ nil)
      Elemento K = primero(listaRest);
      int b = desplaMascara(campo, base, K.clave);
      // desplaMascara(f, r, clave) selecciona el campo f
      // (contando desde la derecha) de clave, según la base r.
      // El resultado, b, es el intervalo 0 ... base−1,
      // y es el número de cubeta para K.
      cubetas[b] = cons(K, cubetas[b]);
      listaRest = resto(listaRest);
   return;
```

```
Lista combinar(Lista[] cubetas, int base)
   // Combina las listas ligadas de todas las cubetas en una sola, L.
   int b; // número de cubeta
   Lista L, cubetaRest;

   L = nil;
   for (b = base−1; b ≥ 0; b --)
      cubetaRest = cubetas[b];
      while (cubetaRest ≠ nil)
        Clave K = primero(cubetaRest);
        L = cons(K, L);
        cubetaRest = resto(cubetaRest);
   return;
```

Análisis y comentarios

Para distribuir una clave es preciso extraer un campo y efectuar unas cuantas operaciones de ligas; el número de pasos está acotado por una constante. Así pues, para todas las claves, `distribuir` efectúa $\Theta(n)$ pasos. Asimismo, `combinar` efectúa $\Theta(n)$ pasos. El número de pasadas de distribución y combinación es `numCampos`, el número de campos empleados para la distribución. Si podemos mantener constante este valor, el número total de pasos efectuados por Ordenamiento por Base será lineal en n.

Nuestra implementación de Ordenamiento por Base usó $\Theta(n)$ espacio extra para los campos de liga, suponiendo que la base está acotada por n. Otras implementaciones que no usan ligas también ocupan un espacio extra en $\Theta(n)$.

Ejercicios

Sección 4.1 Introducción

4.1 Uno de los algoritmos de ordenamiento más fáciles de entender es el que llamamos Maxsort, que funciona como sigue: hallamos la clave más grande, digamos `max`, en la porción no ordenada del arreglo (que en un principio es todo el arreglo) y luego intercambiamos `max` con el elemento que ocupa la última posición de la sección no ordenada. Ahora `max` se considera parte de la porción ordenada, que consiste en las claves más grandes al final del arreglo; ya no está en la sección no ordenada. Este procedimiento se repite hasta que todo el arreglo está ordenado.

a. Escriba un algoritmo para Maxsort suponiendo que un arreglo E contiene n elementos a ordenar, con índices $0, \ldots, n - 1$.

b. ¿Cuántas comparaciones de claves efectúa Maxsort en el peor caso? ¿En promedio?

4.2 A continuación vienen algunos ejercicios acerca de un método de ordenamiento llamado Ordenamiento de Burbuja (*Bubble Sort*), que opera efectuando varias pasadas por el arreglo, comparando pares de claves en posiciones adyacentes e intercambiando sus elementos si no están en orden. Es decir, se comparan las claves primera y segunda y se intercambian si la primera es ma-

yor que la segunda; luego se comparan la (nueva) segunda clave y la tercera y se intercambian si es necesario, y así. Es fácil ver que la clave más grande se irá desplazando hacia el final del arreglo; en pasadas subsiguientes se hará caso omiso de ella. Si en una pasada no se intercambian elementos, el arreglo estará totalmente ordenado y el algoritmo puede parar. El algoritmo que sigue define con más precisión esta descripción informal del método.

Algoritmo 4.14 Ordenamiento de Burbuja

Entradas: *E*, un arreglo de elementos; y $n \geq 0$, el número de elementos.

Salidas: *E* con sus elementos en orden no decreciente según sus claves.

```
void bubbleSort(Elemento[] E, int n)
   int numPares; // el número de pares a comparar
   boolean huboCambio: // true si se efectuó un intercambio
   int j;

   numPares = n - 1;
   huboCambio = true;
   while (huboCambio)
      huboCambio = false;
      for (j = 0; j < numPares; j ++)
        if (E[j] > E[j + 1])
           Intercambiar E[j] y E[j+ 1].
           huboCambio = true;
        // Continuar ciclo for.
      numPares --;
   return;
```

El ejemplo de la figura 4.29 ilustra cómo funciona el Ordenamiento de Burbuja.

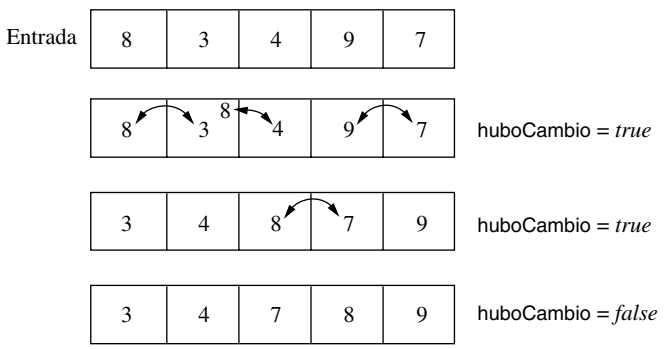


Figura 4.29 Ordenamiento de Burbuja

a. ¿Cuántas comparaciones de claves efectúa Ordenamiento de Burbuja en el peor caso? ¿Qué acomodo de claves constituye un peor caso?
b. ¿Qué acomodo de claves es un caso óptimo para Ordenamiento de Burbuja?, es decir, ¿con qué entradas efectúa el menor número de comparaciones?, ¿cuántas comparaciones efectúa en el mejor caso?

4.3 La corrección de Ordenamiento de Burbuja (ejercicio 4.2) depende de varias cosas. Éstas son fáciles de verificar, y vale la pena hacerlo para reconocer de forma consciente las propiedades matemáticas que intervienen.

a. Demuestre que, después de una pasada por el arreglo, el elemento más grande estará al final.
b. Demuestre que, si ningún par de elementos consecutivos está en desorden, todo el arreglo está ordenado.

4.4 Podemos modificar Ordenamiento de Burbuja (ejercicio 4.2) para que no efectúe comparaciones innecesarias al final del arreglo tomando nota de dónde se efectuó el último intercambio dentro del ciclo `for`.

a. Demuestre que si el último intercambio efectuado en alguna pasada se efectúa en las posiciones j-ésima y $(j + 1)$-ésima, entonces todos los elementos, del $(j + 1)$-ésimo hasta el $(n - 1)$-ésimo, están en su posición correcta. (Observe que esto es más categórico que decir simplemente que estos elementos están en orden.)
b. Modifique el algoritmo de modo que, si el último intercambio efectuado durante una pasada se efectúa en las posiciones j-ésima y $(j + 1)$-ésima, la siguiente pasada no examinará los elementos desde la $(j + 1)$-ésima posición hasta el final del arreglo.
c. ¿Este cambio afecta el comportamiento de peor caso del algoritmo? Si lo hace, ¿cómo lo afecta?

4.5 ¿Se puede hacer algo similar a la mejora del ejercicio anterior para evitar comparaciones innecesarias cuando las claves del principio del arreglo ya están en orden? En tal caso, escriba las modificaciones del algoritmo. Si no se puede, explique por qué.

Sección 4.2 Ordenamiento por inserción

4.6 Observamos que un peor caso de Ordenamiento por inserción es cuando las claves están en un principio en orden descendente. Describa al menos otros dos acomodos iniciales de las claves que también sean peores casos. Presente entradas para las que el número *exacto* de comparaciones de claves (no sólo el *orden asintótico*) sea el peor posible.

4.7 Sugiera un caso óptimo para Ordenamiento por inserción. Describa cómo estarían acomodados los elementos de la lista, e indique cuántas comparaciones de elementos de la lista se efectuarían en ese caso.

4.8 Considere esta variación de Ordenamiento por inserción: con $1 \leq i < n$, para insertar el elemento `E[i]` entre `E[0]` $\leq$ `E[1]` $\leq \cdots \leq$ `E[i - 1]`, se ejecuta Búsqueda Binaria para hallar la posición correcta de `E[i]`.

a. ¿Cuántas comparaciones de claves se efectuarían en el peor caso?
b. ¿Cuántas veces se cambian de lugar elementos en el peor caso?
c. ¿Qué orden asintótico tiene el tiempo de ejecución de peor caso?

d. ¿Se puede reducir el número de traslados colocando los elementos en una lista ligada en vez de un arreglo? Explique.

4.9 En el análisis promedio de Ordenamiento por inserción supusimos que las claves eran distintas. ¿El promedio para todas las posibles entradas, incluidos casos con claves repetidas, sería más alto o más bajo? ¿Por qué?

4.10 Demuestre que una permutación de n elementos tiene cuando más $n(n-1)/2$ inversiones. ¿Cuál(es) permutación(es) tiene(n) exactamente $n(n-1)/2$ inversiones?

4.11 Dé un algoritmo para aplicar Ordenamiento por inserción a una lista ligada de enteros, empleando las operaciones del tipo de datos abstracto `ListaInt` de la sección 2.3.2. Analice sus necesidades de tiempo y espacio. ¿El consumo de espacio depende de si el lenguaje efectúa "recolección de basura" o no? (Vea el ejemplo 2.1.)

Sección 4.3 Divide y vencerás

4.12 Suponga que tiene un algoritmo directo para resolver un problema, el cual efectúa $\Theta(n^2)$ pasos con entradas de tamaño n. Suponga que idea un algoritmo de Divide y vencerás que divide una entrada en dos entradas de la mitad del tamaño y ejecuta $D(n) = n \lg n$ pasos para dividir el problema y $C(n) = n \lg n$ pasos para combinar las soluciones y así obtener una solución para la entrada original. ¿El algoritmo Divide y vencerás es más o menos eficiente que el algoritmo directo? Justifique su respuesta. *Sugerencia:* Vea la ecuación (3.14) y el ejercicio 3.10.

Sección 4.4 Quicksort

4.13 Complete las condiciones posteriores de `extenderRegionChica` del algoritmo 4.3.

4.14 En el algoritmo 4.3, defina la *región media* como el intervalo de índices que contiene los elementos no examinados y la vacante. Para cada una de las líneas 2 a 5 del procedimiento `partir`, ¿qué variables o expresiones variables (podría requerirse algún $+1$ o -1) especifican los extremos izquierdo y derecho de la región media? Para cada una de las líneas 2 a 5, ¿qué extremo de la región media contiene la vacante? La pregunta atañe a la situación inmediatamente *antes* de ejecutarse cada línea.

4.15 ¿Cuántas comparaciones de claves efectúa Quicksort (algoritmos 4.2 y 4.3) si el arreglo ya está ordenado? ¿Cuántos traslados de elementos efectúa?

4.16 Demuestre que si la mejora de "optimación de espacio de pila" de la sección 4.4.4 se usa en el algoritmo 4.2, el tamaño máximo de la pila estará en $O(\log n)$.

4.17 Suponga que, en lugar de escoger `E[primero]` como `pivote`, Quicksort usa como `pivote` la mediana de `E[primero]`, `E[(primero+ultimo)/2]` y `E[ultimo]`. ¿Cuántas comparaciones de claves efectuará Quicksort en el peor caso para ordenar n elementos? (Recuerde contar las comparaciones que se hacen al escoger `pivote`.)

4.18 En este ejercicio se examina un algoritmo alterno para Partir, con código sencillo y elegante. Este método se debe a Lomuto; lo llamamos `partirL`. La idea, que se ilustra en la figura 4.30, consiste en reunir elementos pequeños a la izquierda de la vacante, elementos grandes inmediatamente a la derecha de la vacante y elementos desconocidos (es decir, no examinados) a la extrema derecha del intervalo. En un principio, todos los elementos están en el grupo desconocido. Los elementos "pequeños" y "grandes" se determinan respecto a `pivote`. Cuando `partirL` encuentra un elemento pequeño en el grupo desconocido, lo coloca en la vacante y luego crea una nueva vacante una posición a la derecha transfiriendo un elemento grande desde ese lugar hasta el extremo del intervalo "grande".

```
int partirL(Elemento[] E, Clave pivote, int primero, int ultimo)
   int vacante, desconocido;

1. vacante = primero;
2. for(desconocido = primero + 1; desconocido <= ultimo; desconocido ++)
3.     if(E[desconocido] < pivote)
4.       E[vacante] = E[desconocido];
5.       E[desconocido] = E[vacante+1];
6.       vacante ++;
7. return vacante;
```

En cada iteración de su ciclo, `partirL` compara el siguiente elemento desconocido, que es `E[desconocido]`, con `pivote`. Por último, una vez que todos los elementos se han comparado con `pivote`, se devuelve `vacant` como `puntoPartir`.

a. Al principio de cada una de las líneas de la 2 a la 6, ¿cuáles son las fronteras de la región de claves pequeñas y de la región de claves grandes? Exprese su respuesta utilizando `desconocido` y otras variables de índice.

b. Al principio de la línea 7, ¿cuáles son las fronteras de la región de claves pequeñas y de la región de claves grandes? Exprese su respuesta *sin* usar `desconocido`.

c. ¿Cuántas comparaciones de claves efectúa `partirL` con un subintervalo de E que tiene k elementos? Si Quicksort usa `partirL` en lugar de `partir`, ¿qué impacto tiene ello sobre el número total de comparaciones de claves efectuadas en el peor caso?

4.19 Suponga que el arreglo E contiene las claves 10, 9, 8, . . . , 2, 1, y se debe ordenar con Quicksort.

a. Muestre cómo estarían acomodadas las claves después de las dos primeras invocaciones del procedimiento `partir` del algoritmo 4.3. Indique cuántos traslados de elementos efectúa cada una de estas dos invocaciones de `partir`. A partir de este ejemplo, estime el número total de traslados de elementos que se efectuarían para ordenar n elementos que al principio están en orden decreciente.

b. Haga lo mismo con `partirL`, que se describió en el ejercicio anterior.

c. Cite algunas de las ventajas y desventajas relativas de los dos algoritmos para partir.

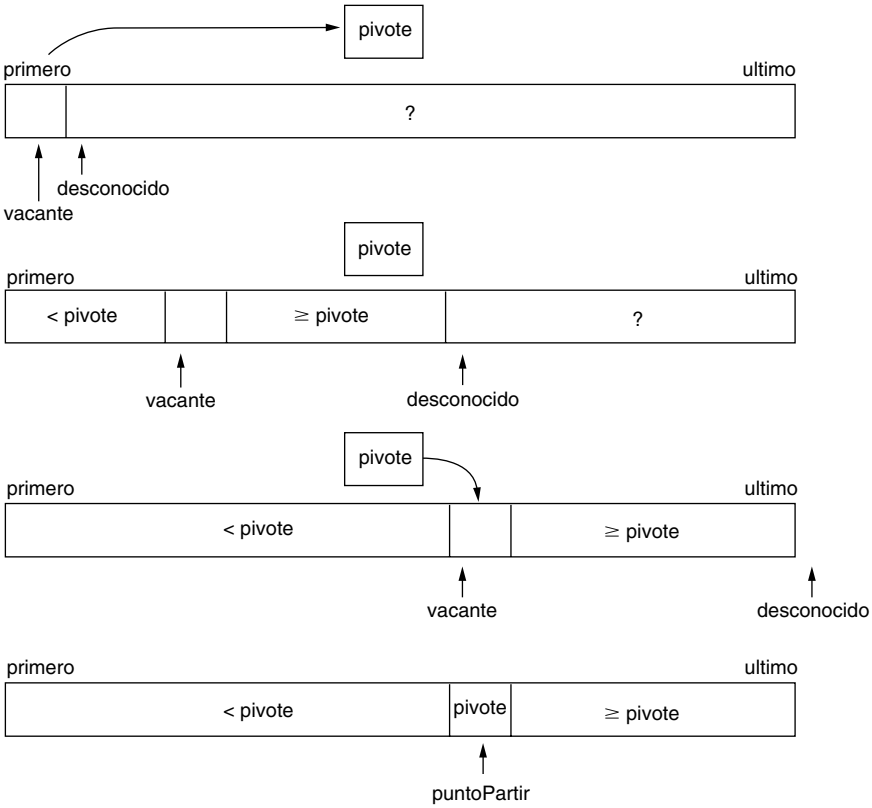


Figura 4.30 Cómo funciona PartirL: vistas inicial, intermedia y final

4.20 Suponga que los *n* elementos del arreglo que se va a ordenar con Quicksort son todos iguales. ¿Cuántas comparaciones de claves efectuará Quicksort para ordenar el arreglo (empleando `partir` en el algoritmo 4.3)? Justifique su respuesta.

4.21 Este ejercicio explora el número medio de traslados de elementos que Quicksort efectúa empleando diferentes versiones de Partir. *Sugerencia para las partes* (a) *y* (b): cuando un elemento se compara con `pivote`, ¿cuál es la probabilidad de que sea necesario cambiarlo de lugar?

a. ¿Cuántos traslados de elementos efectúa Quicksort en promedio si utiliza la subrutina `partirL` del ejercicio 4.18?

b. ¿Cuántos traslados de elementos efectúa Quicksort en promedio si utiliza la subrutina `partir` del algoritmo 4.3?

c. Compare esos resultados con el número de traslados efectuados por Mergesort (vea el ejercicio 4.27).

4.22 Escriba una versión de Quicksort y Partir para listas ligadas de enteros, empleando las operaciones del tipo de datos abstracto `ListaInt` de la sección 2.3.2. Analice sus necesidades de

tiempo y espacio. ¿El consumo de espacio depende de si el lenguaje cuenta o no con "recolección de basura"? (Vea el ejemplo 2.1.)

Sección 4.5 Fusión de sucesiones ordenadas

4.23 Dé un algoritmo para fusionar dos listas ligadas ordenadas de enteros, empleando las operaciones del tipo de datos abstracto `ListaInt` de la sección 2.3.2.

★ **4.24** Suponga que los subintervalos de arreglo que se van a fusionar tienen longitudes k y m, donde k es mucho menor que m. Describa un algoritmo de fusión que aproveche esto para efectuar cuando más (digamos) $(k + m)/2$ comparaciones, siempre que k sea lo bastante pequeño en relación con m. ¿Qué tan pequeño tiene que ser k para alcanzar esta cota en el peor caso? ¿Existe un intervalo de k para el cual pueda lograrse la cota $\sqrt{k + m}$? ¿Qué puede usted decir acerca del número de traslados de elementos que se requiere en estos casos?

4.25 Demuestre que el número de permutaciones que pueden formarse fusionando dos segmentos ordenados A y B con longitudes k y m, donde $k + m = n$, es $\binom{n}{k} = \binom{n}{m}$. (Recuerde esta notación de la ecuación 1.1.) Suponga que $k \leq m$ para precisar más la demostración, y que no hay claves repetidas. *Sugerencia:* Formule una recurrencia basada en la relación entre `A[0]` y `B[0]`, y luego estudie el ejercicio 1.2. Hay varias otras formas de enfocar este problema que también funcionan.

Sección 4.6 Mergesort

4.26 ¿Cuántas comparaciones de claves efectúa Mergesort si las claves ya están en orden cuando se inicia el ordenamiento?

4.27 Describimos Mergesort (algoritmo 4.5) suponiendo que Fusionar desarrollaba su salida en un arreglo de trabajo y al terminar copiaba el contenido de ese arreglo en el arreglo de entrada.

a. Deduzca una estrategia para cambiar del arreglo de entrada al de trabajo y viceversa de modo que se evite ese copiado extra. Es decir, en niveles alternos de la recursión, el arreglo de entrada original tiene los datos a fusionar o bien el arreglo de trabajo los tiene.

b. Con la optimación anterior, ¿cuántos traslados de elementos efectúa Mergesort en promedio? Compare esa cifra con la de Quicksort (vea el ejercicio 4.21).

4.28 Escriba una versión de Mergesort para listas ligadas de enteros, empleando las operaciones del tipo de datos abstracto `ListaInt` de la sección 2.3.2. Analice sus necesidades de tiempo y espacio. ¿El consumo de espacio depende de que el lenguaje cuente o no con "recolección de basura"? (Vea el ejemplo 2.1.)

4.29 Para el análisis de Mergesort empleando árbol de recursión (sección 4.6), donde D es la profundidad máxima del árbol y B es el número de casos base que hay en la profundidad $D - 1$, verifique que $B = 2^D - n$.

4.30 Deduzca el valor mínimo de la expresión $(\alpha - \lg \alpha)$ dentro del intervalo (1, 2), que se usó en la demostración del teorema 4.6. Demuestre que es $(1 + \ln \ln 2)/\ln 2$.

Sección 4.7 Cotas inferiores para ordenar comparando claves

4.31 Dibuje el árbol de decisión para Quicksort con $n = 3$. (Tendrá que modificar las convenciones un poco. Algunas ramas deberán rotularse "$\leq$" o "$\geq$".)

4.32

a. Dé un algoritmo para ordenar cuatro elementos empleando sólo cinco comparaciones de claves en el peor caso.

★ **b.** Dé un algoritmo para ordenar cinco elementos que sea óptimo en el peor caso. (Volveremos a este problema en el capítulo 5, después de introducir ciertas técnicas nuevas.)

★ **4.33** Utilizando el resultado del ejercicio 4.25, dé una cota inferior basada en árboles de decisión para el número de combinaciones que se necesitan para fusionar dos segmentos ordenados de longitudes k y m, donde $k + m = n$. Suponga que $k \leq m$ para hacer más definido su análisis, y que no hay claves repetidas. En su expresión pueden intervenir tanto k como m, pues no se supone que sean iguales; también podría usarse n por comodidad, pero puede sustituirse por $k + m$.

a. Primero, deduzca una expresión que podría incluir sumatorias, pero que sea exacta.

b. Para $k = m = n/2$, obtenga una aproximación en forma cerrada que sea cercana, pero siempre menor que, su expresión de la parte (a). ("Forma cerrada" implica que la expresión no debe contener sumatorias ni integrales.) Compare su expresión con el teorema 4.4. ¿Qué diferencias hay?

★★ **c.** Esta parte podría requerir matemáticas un tanto complicadas. Obtenga una aproximación en forma cerrada para el caso en que $k < m$. Al igual que en la parte (b), deberá ser cercana, pero siempre menor, que su expresión de la parte (a). Para obtener buenos resultados, podría ser conveniente considerar varios intervalos para la relación entre k y n.

Sección 4.8 Heapsort

4.34 Suponga que los elementos de un arreglo son (comenzando con el índice 1) 25, 19, 15, 5, 12, 4, 13, 3, 7, 10. ¿Este arreglo representa un montón?, justifique su respuesta.

4.35 Suponga que el arreglo a ordenar (en orden alfabético) con Heapsort contiene inicialmente la siguiente sucesión de letras:

C O M P L E J I D A D

Muestre cómo quedarían acomodadas en el arreglo después de la fase de construcción del montón (algoritmo 4.7). ¿Cuántas comparaciones de claves se efectúan para construir el montón con estas claves?

4.36 Los nodos de un montón se almacenan en un arreglo E nivel por nivel comenzando con la raíz, y de izquierda a derecha dentro de cada nivel. Demuestre que el hijo izquierdo del nodo que está en la i-ésima celda está en la $2i$-ésima celda. (Recuerde que un montón se almacena con la raíz en `E[1]`. No se usa `E[0]`.)

4.37 Un arreglo con claves distintas y en orden decreciente se va a ordenar (en orden creciente) con Heapsort (no Heapsort Acelerado).

a. ¿Cuántas comparaciones de claves se efectúan en la fase de construcción del montón (algoritmo 4.7) si hay 10 elementos?

b. ¿Cuántas se efectúan si hay n elementos? Muestre cómo dedujo su respuesta.

c. ¿Un arreglo en orden decreciente es un caso óptimo, un peor caso o un caso intermedio para el algoritmo 4.7? Justifique su respuesta.

4.38 Heapsort, tal como se describe en el texto, no es exactamente un ordenamiento en su lugar porque la recursión ocupa espacio en la pila de marcos de activación.

a. ¿Cuánto espacio se ocupa en la pila de marcos de activación?

b. Convierta `repararMonton` en un procedimiento iterativo.

c. Convierta `repararMontonRapido` en un procedimiento iterativo.

d. Convierta `construirMonton` en un procedimiento iterativo invocando `repararMonton-Rapido` (o `repararMonton`) dentro de un ciclo **`for`** que parte de `E[n/2]` y retrocede hasta `E[1]`, la raíz del montón.

e. ¿Cuántas comparaciones efectúa la versión iterativa de `construirMonton` en el peor caso (qué orden asintótico tiene)?

4.39 Este ejercicio presenta un argumento alterno para el análisis de peor caso de la fase de construcción de montón de Heapsort. El procedimiento `construirMonton` invoca a `repararMonton` una vez por cada nodo del montón, y sabemos que el número de comparaciones de claves que hace `repararMonton` en el peor caso es dos veces la altura del nodo. (Recuerde que la *altura* de un nodo de un árbol binario es la altura del subárbol cuya raíz es ese nodo.) Así pues, el número de comparaciones efectuadas en el peor caso es cuando más la suma de las alturas de todos los nodos. Demuestre que la suma de las alturas de los nodos de un montón que tiene n nodos es cuando más $n - 1$. *Sugerencia:* Utilice una estrategia de tachado, tachando sistemáticamente una rama del árbol por cada unidad de altura de la sumatoria.

4.40 Podríamos eliminar una invocación de `repararMonton` en Heapsort (algoritmo 4.8) cambiando el control del ciclo **`for`** a

```
for (tamaño = n; tamaño >= 3; tamaño – –;)
```

¿Qué enunciado, si acaso, tendría que agregarse después del ciclo **`for`** para ocuparse del caso en que quedan dos elementos en el montón? ¿Cuántas comparaciones se eliminan, si es que se elimina alguna?

4.41 Suponga que tiene un montón con `tamaño` elementos almacenados en un arreglo H, y que quiere agregar un elemento nuevo K. Utilizando `subirMonton` de la sección 4.8.6, el procedimiento es simplemente

```
int insertarMonton(Elemento[] H, Elemento K, int tamaño)
    int nuevoTamaño = tamaño + 1;
    subirMonton(K, 1, K, nuevoTamaño);
    return nuevoTamaño;
```

a. ¿Cuántas comparaciones de claves efectúa `insertarMonton` en el peor caso con un montón que contiene n elementos después de la inserción?

b. Una versión anterior de Heapsort utilizaba `insertarMonton` para construir un montón a partir de los elementos a ordenar insertando los elementos, uno por uno, en un montón que en un principio estaba vacío. ¿Cuántas comparaciones efectúa este método en el peor caso para construir un montón de n elementos?

c. ¿Cuántas comparaciones realizaría Heapsort en el peor caso si usara `insertarMonton` para construir el montón, como se describe en la parte (b)?

4.42 Un montón contiene 100 elementos, que por casualidad están en orden decreciente en el arreglo, con claves 100, 99, . . . , 1.

a. Muestre cómo funcionaría `borrarMax` con este montón (para eliminar sólo la clave 100) si se implementa con `repararMontonRapido`. Muestre específicamente las comparaciones y traslados de elementos que se realizan.

b. ¿Cuántas comparaciones se efectúan?

c. ¿Hace más o menos comparaciones la implementación que usa `repararMonton`?

4.43 Un arreglo de claves distintas en orden decreciente se ordenará (en orden creciente) empleando Heapsort Acelerado. Suponga que se usa `repararMontonRapido` en lugar de `repararMonton`.

a. ¿Cuántas comparaciones de claves se hacen en la fase de construcción del montón (algoritmo 4.7) si hay 31 elementos?

b. ¿Cuántas se efectúan si hay n elementos? Explique cómo obtuvo su respuesta.

c. ¿Un arreglo en orden decreciente es un caso óptimo, un peor caso o un caso intermedio para el algoritmo 4.7, empleando `repararMontonRapido`? Justifique su respuesta.

⋆ **4.44** Demuestre que $\lceil \lg(\lfloor \frac{1}{2} h \rfloor + 1) \rceil + 1 = \lceil \lg(h + 1) \rceil$ para todos los enteros $h \geq 1$.

Sección 4.10 Shellsort

4.45 Suponga que se usan cinco incrementos en Shellsort y que todos son constantes (independientes de n, el número de elementos a ordenar). Demuestre que, aunque el número de comparaciones que se efectúan en el peor caso podría ser un poco menor que el número de comparaciones efectuadas por Ordenamiento por inserción, sigue estando en $\Theta(n^2)$.

Sección 4.11 Ordenamiento por base

4.46 Suponga que Ordenamiento por base efectúa m pasadas de distribución con claves de w bits (donde m es un divisor de w) y que hay una cubeta por cada patrón de w/m bits, de modo que `base` $= 2^{w/m}$. Puesto que se efectúan mn distribuciones de claves, podría parecer provechoso reducir m. ¿Qué tan grande debería ser la nueva `base` si m se reduce a la mitad?

Problemas adicionales

4.47 Suponga que un algoritmo efectúa m^2 pasos con un arreglo de m elementos (para cualquier $m \geq 1$). El algoritmo se aplicará a dos arreglos, A_1 y A_2 (por separado). Los arreglos contienen un total de n elementos. A_1 tiene k elementos y A_2 tiene $n - k$ elementos ($0 \leq k \leq n$).

¿Con qué valor(es) de k se trabaja más? ¿Con qué valor(es) de k se trabaja menos? Justifique sus respuestas. (Recuerde que un ejemplo no es una demostración. Hay una buena solución de este problema que usa cálculo sencillo.)

4.48 Ordenar o no ordenar: bosqueje un método razonable para resolver cada uno de los problemas siguientes. Dé el orden de la complejidad de peor caso de sus métodos.

a. Se le entrega una pila de miles de cuentas telefónicas y miles de cheques que los consumidores han enviado para pagar sus cuentas (suponga que los cheques llevan el número telefónico). Averigüe quién no pagó.

b. Se le proporciona un arreglo en el que cada elemento contiene el título, el autor, el número de llamada y la casa editorial de todos los libros de una biblioteca escolar, y otro arreglo de 30 casas editoriales. Averigüe cuántos de los libros publicó cada una de esas compañías.

c. Se le da un arreglo que contiene expedientes de préstamo de todos los libros solicitados en préstamo a la biblioteca universitaria durante el año anterior. Determine cuántas personas distintas solicitaron al menos un libro en préstamo.

★ **4.49** Resuelva la ecuación de recurrencia siguiente:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + cn \qquad \text{para } n > 2 \qquad T(2) = 1$$

donde c es alguna constante positiva.

4.50 Dé un algoritmo eficiente que trabaje en su lugar para reacomodar un arreglo de n elementos de modo que todas las claves negativas estén antes de todas las claves no negativas. ¿Cuál es la rapidez de su algoritmo?

4.51 Un método de ordenamiento es *estable* si en la sucesión ordenada claves iguales están en el mismo orden relativo que tenían en la sucesión original. (Es decir, un ordenamiento es estable si, para cualquier $i < j$ tal que al principio `E[i]` = `E[j]`, el ordenamiento traslada `E[i]` a `E[k]` y `E[j]` a `E[m]` para alguna k y alguna m tal que $k < m$.) ¿Cuáles de los algoritmos siguientes son estables? Para cada uno de los que no son estables, dé un ejemplo en el que se altere el orden relativo de dos claves iguales.

a. Ordenamiento por inserción.

b. Maxsort (ejercicio 4.1).

c. Ordenamiento de Burbuja (ejercicio 4.2).

d. Quicksort.

e. Heapsort.

f. Heapsort Acelerado.

g. Shellsort.

h. Ordenamiento por base.

4.52 Suponga que tiene un arreglo de 1,000 expedientes en el que sólo unos cuantos están en desorden, aunque no están muy lejos de su posición correcta. ¿Qué algoritmo de ordenamiento usaría para ordenar todo el arreglo? Justifique su decisión.

4.53 ¿Qué algoritmo de ordenamiento descrito en este capítulo sería difícil de adaptar al ordenamiento de elementos almacenados en una lista ligada (sin cambiar el orden asintótico de peor caso)?

4.54 En casi todas las partes de este capítulo hemos supuesto que las claves del conjunto a ordenar son distintas. Es común que haya claves repetidas. Tal repetición podría facilitar el ordenamiento, pero es posible que los algoritmos diseñados para claves distintas (o en su mayor parte distintas) no aprovechen la repetición. Consideremos el caso extremo en el que las claves sólo pueden tener dos valores, 0 y 1.

a. ¿Qué orden asintótico tiene el número de comparaciones efectuadas por Ordenamiento por inserción en el peor caso? (Describa una entrada de peor caso.)

b. ¿Qué orden tiene el número de comparaciones de claves efectuadas por Quicksort en el peor caso? (Describa una entrada de peor caso.)

c. Dé un algoritmo eficiente para ordenar un conjunto de n elementos cuyas claves podrían ser 0 o 1. ¿Qué orden tiene el tiempo de ejecución de peor caso de su algoritmo?

⋆ **4.55** Cada uno de los n elementos de un arreglo puede tener uno de los valores de clave *rojo*, *blanco* o *azul*. Sugiera un algoritmo eficiente para reacomodar los elementos de modo que todos los *rojo*s estén antes de todos los *blanco*s, y todos los *blanco*s estén antes de todos los *azul*es. (Podría darse el caso de que no haya elementos de uno o dos de los colores.) Las únicas operaciones que pueden efectuarse con los elementos son examinar una clave para averiguar qué color es, e intercambiar dos elementos (especificados por sus índices). ¿Qué orden asintótico tiene el tiempo de ejecución de peor caso de su algoritmo? (Existe una solución lineal.)

4.56 Suponga que tiene una computadora con n posiciones de memoria, numeradas de 1 a n, y una instrucción CIC, llamada "comparar-intercambiar". Para $1 \leq i, j \leq n$, CIC i, j compara las claves que están en las celdas de memoria i y j y las intercambia si es necesario de modo que la clave más pequeña esté en la celda de índice más pequeño. La instrucción CIC puede servir para ordenar. Por ejemplo, el programa que sigue ordena con $n = 3$:

```
CIC 1,2
CIC 2,3
CIC 1,2
```

a. Escriba un programa eficiente empleando sólo instrucciones CIC para ordenar seis elementos. (*Sugerencia:* Escriba programas para $n = 4$ y $n = 5$ primero. Es fácil escribir programas para $n = 4$, 5 y 6 empleando 6, 10 y 15 instrucciones, respectivamente. Sin embargo, ninguno de ellos es óptimo.)

b. Escriba un programa CIC para ordenar n elementos en n celdas para una n fija pero arbitraria. Utilice el mínimo de instrucciones que pueda. Describa la estrategia empleada por su programa e incluya comentarios en los puntos apropiados. Puesto que no hay instrucciones de ciclos ni de prueba, se pueden usar puntos suspensivos para indicar la repetición de instrucciones de cierta forma; por ejemplo:

```
CIC 1,2
CIC 2,3
⋮
CIC n − 1, n
```

c. ¿Cuántas instrucciones CIC tiene el programa que escribió para la parte (b)?

d. Dé una cota inferior para el número de instrucciones CIC que se necesitan para ordenar n elementos.

4.57

a. Suponga que es posible ejecutar simultáneamente instrucciones CIC (descritas en el ejercicio anterior) si están operando con claves que están en diferentes celdas de memoria. Por ejemplo, CIC 1,2, CIC 3,4, CIC 5,6, etc., se pueden ejecutar al mismo tiempo. Escriba un algoritmo para ordenar cuatro elementos en sólo tres unidades de tiempo. (Recuerde que ordenar cuatro elementos requiere cinco comparaciones.)

★ **b.** Escriba un algoritmo que use instrucciones CIC (simultáneas) para ordenar n elementos en $o(\log(n!))$ unidades de tiempo.

★ **4.58** M es una matriz de $n \times n$ enteros en la que los elementos de cada fila están en orden creciente (leyendo de izquierda a derecha) y los elementos de cada columna están en orden creciente (leyendo de arriba hacia abajo). Escriba un algoritmo eficiente para encontrar la posición de un entero x en M, o determinar que x no está ahí. Indique cuántas comparaciones de x con elementos de la matriz efectúa su algoritmo en el peor caso. Puede usar comparaciones de tres vías, es decir, una comparación de x con `M[i][j]` dice si $x <$ `M[i][j]`, $x =$ `M[i][j]` o $x >$ `M[i][j]`.

★ **4.59** E es un arreglo que contiene n enteros y queremos obtener la sumatoria máxima para una subsucesión contigua de elementos de E. (Si todos los elementos de una sucesión son negativos, definimos la subsucesión máxima contigua como la sucesión vacía con sumatoria igual a cero.) Por ejemplo, consideremos la sucesión

$$38, -62, 47, -33, 28, 13, -18, -46, 8, 21, 12, -53, 25.$$

La *sumatoria de subsucesión máxima* para este arreglo es 55. La subsucesión contigua máxima se da en las posiciones 3 a 6 (inclusive).

a. Escriba un algoritmo que halle la sumatoria de subsucesión máxima en un arreglo. ¿Qué orden asintótico tiene el tiempo de ejecución de su algoritmo? (Los datos de las tablas 1.1 y 1.2 provienen de diversos algoritmos para este problema. Como indican esas tablas, hay muchas soluciones de complejidad variable, incluida una lineal.)

b. Demuestre que cualquier algoritmo para este problema debe examinar todos los elementos del arreglo en el peor caso. (De modo que cualquier algoritmo ejecuta $\Omega(n)$ pasos en el peor caso.)

★ **4.60** En lugar de reacomodar un arreglo E de expedientes grandes durante el ordenamiento, es fácil modificar el código para trabajar con un arreglo de *índices* de dichos expedientes y reacomodar los índices. Una vez ordenado, el arreglo de índices definirá la permutación correcta, π, del arreglo original, E, para que sus expedientes queden ordenados. Es decir, $E[\pi[0]]$ es el expediente mínimo, $E[\pi[1]]$ es el siguiente en orden creciente, etc. Este ejercicio estudia el problema de reacomodar los expedientes mismos, una vez determinada la permutación correcta.

Su algoritmo recibe E, un arreglo de expedientes y un entero n, tal que están definidos elementos de E para los índices $0, 1, \ldots, n-1$. El algoritmo también recibe otro arreglo, π, en el que se almacena una permutación de los números $0, 1, \ldots, n-1$.

a. Escriba un algoritmo que reacomode los expedientes de E en el orden $\pi[0], \pi[1], \ldots, \pi[n-1]$. Es decir, el expediente que originalmente estaba en $E[\pi[0]]$ deberá quedar en $E[0]$, el expediente que originalmente estaba en $E[\pi[1]]$ deberá quedar en $E[1]$, y así sucesivamente. Suponga que los expedientes de E son grandes; en particular, no cabrían en el arreglo π. Su algoritmo puede destruir π, y puede almacenar valores fuera del intervalo $0, \ldots, n-1$ en los elementos de π. Si usa espacio extra, especifique cuánto.

b. ¿Cuántas veces en total cambia expedientes de lugar su algoritmo en el peor caso? ¿El tiempo de ejecución de su algoritmo es proporcional al número de traslados? De ser así, explique por qué. Si no, ¿cuál es el orden asintótico del tiempo de ejecución?

4.61 ¿Qué método de ordenamiento usaría para cada uno de los problemas siguientes? Explique su decisión.

a. Una universidad del sur de California tiene cerca de 30,000 estudiantes de tiempo completo y cerca de 10,000 estudiantes de tiempo parcial. (No se permite la inscripción de más de 50,000 estudiantes a la vez en la universidad, debido a limitaciones de estacionamiento de vehículos.) Cada expediente de estudiante contiene el nombre del estudiante, un número de identificación de nueve dígitos, su dirección, sus calificaciones, etc. Un nombre se almacena como una cadena de 41 caracteres, 20 caracteres para el nombre de pila, 20 para el apellido, y uno para la inicial intermedia.

El problema consiste en producir una lista de alumnos en orden alfabético para cada uno de los aproximadamente 5,000 cursos al principio de cada semestre. Estas listas se entregan a los profesores antes del primer día de clases. El tamaño máximo de un grupo es de 200. La mayor parte de las clases tiene aproximadamente 30 estudiantes. Las entradas para cada clase son un arreglo no ordenado con cuando más 200 expedientes. Dichos expedientes contienen el nombre de un estudiante, su número de identificación, su estatus universitario (freshman, sophomore, junior, senior, graduado), así como la dirección del expediente completo del estudiante en disco.

b. El problema consiste en ordenar 500 exámenes alfabéticamente según el apellido del estudiante. Una persona ordenará los exámenes, para ello dispone de una oficina con dos escritorios que temporalmente se han despejado de otros papeles, libros y tazas para café. Es la 1:00 A.M. y la persona quisiera irse a casa lo antes posible.

4.62 ¿Siempre se cumple que un arreglo que ya está ordenado es una entrada de caso óptimo para los algoritmos de ordenamiento? Presente un argumento o un contraejemplo.

4.63 Suponga que tiene un arreglo no ordenado A con n elementos y quiere saber si el arreglo contiene elementos repetidos.

a. Bosqueje (claramente) un método eficiente para resolver este problema.

b. ¿Qué orden asintótico tiene el tiempo de ejecución de su método en el peor caso? Justifique su respuesta.

c. Suponga que sabe que los n elementos son enteros del intervalo $1, \ldots, 2n$, por lo que pueden efectuarse otras operaciones además de comparar claves. Sugiera un algoritmo para el mismo problema pero especializado para aprovechar esta información. Indique el orden asintótico del tiempo de ejecución de peor caso para esta solución. Dicho orden deberá ser más bajo que el de su solución para la parte (a).

4.64 Suponga que se mantiene un arreglo grande con la política siguiente. En un principio la lista está ordenada. Cuando se agregan elementos nuevos, se anexan al final del arreglo y se cuentan. Cada vez que el número de elementos nuevos llega a 10, el arreglo vuelve a ordenarse y el contador se pone en ceros. ¿Qué estrategia sería bueno usar para reordenar el arreglo? ¿Por qué?

4.65 Se le da un arreglo ordenado `E1` con k elementos y un arreglo no ordenado `E2` con $\lg k$ elementos. El problema consiste en combinar los arreglos en un solo arreglo ordenado (con n elementos, donde $n = k + \lg k$). Puede suponer que el primer arreglo tiene espacio para los n elementos, si lo desea.

Una forma de resolver el problema es simplemente ordenar el arreglo combinado, efectuando $\Theta(n \log n)$ comparaciones de claves en el peor caso. Queremos algo mejor. Describa otros dos algoritmos para este problema. (No tiene que escribir código.) Describa claramente los pasos principales de cada método (con suficiente detalle como para poder estimar fácilmente el número de comparaciones de claves que se efectúan). Indique el orden asintótico del número de comparaciones de claves para cada método en función de n (preferible) o en función de k (si es difícil expresarlo en función de n).

Para al menos uno de los métodos que describa, el número de comparaciones de claves deberá estar en $o(n \log n)$.

4.66 En una universidad grande se debe ejecutar un programa cada semestre para detectar las cuentas de estudiantes en el sistema de cómputo que deben darse de baja. Cualquier estudiante inscrito puede tener una cuenta, y cualquiera tiene un "periodo de gracia" de un semestre después de salir de la universidad. Por tanto, una cuenta debe darse de baja si el estudiante no está inscrito actualmente y no estuvo inscrito en el semestre anterior.

a. Bosqueje un algoritmo para elaborar una lista de las cuentas que deben darse de baja. Los dos párrafos que siguen describen los archivos con los que se va a trabajar. No es preciso escribir código, pero debe quedar bien claro lo que se está haciendo.[1]

El Archivo de Cuentas está ordenado por nombre de usuario. Cada elemento contiene el nombre de usuario, el nombre real, el número de identificación, la fecha de creación, la fecha de expiración, el código de carrera y otros campos. Se asigna una fecha de expiración del 31 de diciembre de 2030 cuando se establece la cuenta de un estudiante porque en ese momento se desconoce la verdadera fecha de expiración. En las cuentas de profesores y otras cuentas no de estudiantes el campo de número de identificación contiene cero. Hay aproximadamente 12,000 cuentas en el archivo.

El Archivo Maestro de Estudiantes, mantenido por la administración, contiene un expediente para cada estudiante que está inscrito actualmente (aproximadamente 30,000 expedientes). Está ordenado alfabéticamente por nombre real. Cada expediente incluye el número de identificación del estudiante y otra información. Hay nombres repetidos; es decir, hay ocasiones en que diferentes estudiantes tienen el mismo nombre. Se tiene acceso al Archivo Maestro de Estudiantes para el semestre anterior.

b. Sea n el número de cuentas y s el número de estudiantes. Exprese el orden asintótico del tiempo de ejecución de su método en términos de n y s. (Justifique su respuesta.)

[1] El administrador del sistema de cómputo de una universidad real informó que dos personas escribieron programas para este problema. Uno tardó 45 minutos en ejecutarse; el otro tardó 2 minutos.

c. Los problemas del mundo real a menudo tienen complicaciones. Describa al menos una situación que podría presentarse (y sea razonablemente verosímil) pero no esté cubierta claramente en las especificaciones.

Programas

Para cada programa, incluya un contador que cuente comparaciones de claves. Incluya entre sus datos de prueba archivos en los que las claves estén en orden decreciente, en orden creciente y en orden aleatorio. Utilice archivos con distintos números de elementos. Las salidas deben incluir el número de elementos y el número de comparaciones efectuadas.

1. Quicksort. Use las mejoras descritas en la sección 4.4.4.
2. Heapsort Acelerado. Muestre el montón completo una vez que se han insertado todos los elementos.
3. Ordenamiento por Base.
4. Mergesort. Implemente la mejora sugerida en el ejercicio 4.27.

Notas y referencias

Gran parte del material de este capítulo se basa en Knuth (1998), sin duda la principal referencia en materia de ordenamiento y problemas afines. Se recomienda sobremanera a los lectores interesados consultar este libro en busca de más algoritmos, análisis, ejercicios y referencias. Entre las fuentes originales de los algoritmos están: Hoare (1962) para Quicksort, incluidas variaciones y aplicaciones; Williams (1964) para Heapsort (con una de las primeras mejoras sugerida por Floyd (1964)) y Shell (1959) para Shellsort.

La versión de Partir dada en el algoritmo 4.3 es muy cercana a la publicada por Hoare. Pruebas empíricas recientes (no publicadas) han demostrado que las "optimaciones" que reducen el número de instrucciones en el ciclo interior, a expensas de instrucciones adicionales en otros puntos, resultan contraproducentes en las estaciones de trabajo modernas. Ello al parecer se debe a que la instrucción eliminada, una comparación de dos índices, se efectúa en registros de máquina, así que de todos modos es muy rápida, mientras que las instrucciones adicionales implican accesos a la memoria y tardan relativamente más.

La versión del procedimiento Partir del ejercicio 4.18 aparece en Bentley (1986), donde se atribuye a N. Lomuto.

Al parecer, Carlsson (1987) es el primer trabajo en describir una versión de Heapsort que usa aproximadamente $n \lg(n)$ comparaciones en lugar de aproximadamente $2n \lg(n)$, en el peor caso. Varios investigadores redescubrieron la idea posteriormente.

El argumento conciso presentado en la sección 4.2 para el número medio de inversiones en una permutación fue sugerido por Sampath Kannan. Al final de la sección 4.8 comentamos que hay algoritmos que efectúan menos comparaciones que Mergesort en el peor caso. El algoritmo Ford-Johnson, llamado Fusión-Inserción, es de ese tipo. Se sabe que es óptimo con valores pequeños de n. Inserción Binaria es otro algoritmo que efectúa aproximadamente $n \lg n$ comparaciones en el peor caso. En Knuth (1998) se describen estos algoritmos, se analizan varias opciones de incrementos para Shellsort, se demuestra el teorema 4.16 y se analiza el ordenamiento externo.

El problema de ordenamiento del ejercicio 4.55 se resuelve en Dijkstra (1976) donde se le llama "Problema de la Bandera Nacional Holandesa". Bentley (1986) presenta algo de historia y varias soluciones del problema de la sumatoria de subsecuencia máxima (ejercicio 4.59). Los datos de la tabla 1.2 y todas las columnas de la tabla 1.1 menos la exponencial provienen de soluciones a este problema. El ejercicio 4.61 fue una aportación de Roger Whitney.

Los procedimientos de ordenamiento adaptativos aprovechan permutaciones favorables de las entradas para ordenar con mayor eficiencia. Estivill-Castro y Wood (1996) estudian este tema a fondo.

5

Selección y argumentos de adversario

5.1 Introducción

En este capítulo estudiaremos varios problemas que se pueden agrupar bajo el nombre general de *selección*. Un ejemplo muy conocido es determinar qué elemento es la mediana de un conjunto. Además de encontrar algoritmos para resolver los problemas de forma eficiente, exploraremos las *cotas inferiores* de los problemas. Presentaremos una técnica de amplia aplicación, llamada *argumentos de adversario*, para establecer cotas inferiores.

5.1.1 El problema de la selección

Supóngase que E es un arreglo que contiene n elementos con claves de algún conjunto ordenado linealmente y sea k un entero tal que $1 \leq k \leq n$. El problema de *selección* es el problema de hallar en E el elemento con la k-ésima llave más pequeña. Decimos que tal elemento *tiene rango k*. Al igual que con la mayor parte de los elementos de ordenamiento que estudiamos, supondremos que las únicas operaciones que pueden efectuarse con las claves son comparaciones de pares de claves (y copiar o cambiar de lugar elementos). En este capítulo consideraremos idénticas las claves y los elementos, porque nos concentraremos en el número de comparaciones de claves, olvidándonos por lo regular del traslado de elementos. Además, al almacenar claves en un arreglo, usaremos las posiciones $1, \ldots, n$, en congruencia con la terminología de rangos común, en lugar de $0, \ldots, n-1$. La posición 0 del arreglo simplemente no se usa.

En el capítulo 1 resolvimos el problema de selección para el caso $k = n$, pues ese problema no es sino la búsqueda de la clave más grande. Examinamos un algoritmo directo que efectuaba $n-1$ comparaciones de claves y demostramos que ningún algoritmo podía efectuar menos. El caso dual de $k = 1$, es decir, hallar la clave más pequeña, se puede resolver de forma similar.

Otro caso muy común del problema de selección es cuando $k = \lceil n/2 \rceil$, es decir, cuando queremos hallar el elemento de en medio o *mediana*. La mediana es útil para interpretar conjuntos muy grandes de datos, como los ingresos de todos los habitantes de un país dado o de quienes ejercen cierta profesión, el precio de las casas o los puntajes en los exámenes de admisión universitarios. En lugar de incluir todo el conjunto de datos, los informes noticiosos, por ejemplo, lo resumen presentando la media (promedio) o la mediana. Es fácil calcular el promedio de n números en tiempo $\Theta(n)$. ¿Cómo podemos calcular la mediana de forma eficiente?

Desde luego, todos los casos del problema de selección se pueden resolver ordenando a E; entonces, sea cual sea el rango k que nos interese, $E[k]$ sería la respuesta. El ordenamiento requiere $\Theta(n \log n)$ comparaciones de claves y acabamos de observar que, con algunos valores de k, el problema de selección se puede resolver en tiempo lineal. Intuitivamente, hallar la mediana parece ser el caso más difícil del problema de selección. ¿Podemos hallar la mediana en tiempo lineal? ¿o podemos establecer una cota inferior para la localización de la mediana que sea más que lineal, quizá $\Theta(n \log n)$? Contestaremos estas preguntas en el presente capítulo y delinearemos un algoritmo para el problema de selección general.

5.1.2 Cotas inferiores

Hasta aquí hemos usado el árbol de decisión como principal técnica para establecer cotas inferiores. Recordemos que los nodos internos del árbol de decisión de un algoritmo representan las comparaciones que el algoritmo realiza y las hojas representan las salidas. (En el caso del problema de búsqueda de la sección 1.6, los nodos internos también representaban salidas.) El número de comparaciones hechas en el peor caso es la altura del árbol; la altura es por lo menos $\lceil \lg L \rceil$, donde L es el número de hojas.

En la sección 1.6 utilizamos árboles de decisión para obtener la cota inferior de peor caso de $\lceil \lg(n+1) \rceil$ para el problema de búsqueda. Ése es exactamente el número de comparaciones que efectúa Búsqueda Binaria, así que un argumento de árbol de decisión nos dio la mejor cota inferior posible. En el capítulo 4 usamos árboles de decisión para obtener una cota inferior de $\lceil \lg n! \rceil$ o aproximadamente $\lceil n \lg n - 1.5\, n \rceil$, para el ordenamiento. Hay algoritmos cuyo desempeño es muy cercano a esta cota inferior por lo que, una vez más, un argumento de árbol de decisión dio un resultado muy sólido. Sin embargo, los argumentos de árbol de decisión no funcionan muy bien con el problema de selección.

Un árbol de decisión para el problema de selección debe tener por lo menos n hojas porque cualquiera de las n claves del conjunto podría ser la salida, es decir, el k-ésimo elemento más pequeño. Por ello, podemos concluir que la altura del árbol (y el número de comparaciones efectuadas en el peor caso) es por lo menos $\lceil \lg n \rceil$. Sin embargo, ésta no es una buena cota inferior; ya sabemos que incluso el caso fácil de hallar la clave más grande requiere por lo menos $n - 1$ comparaciones. ¿Qué error tiene el argumento de árbol de decisión? En un árbol de decisión para un algoritmo que halla la clave más grande, algunas salidas aparecen en más de una hoja, y de hecho habrá más de n hojas. Para ver esto, el ejercicio 5.1 pide al lector dibujar el árbol de decisión para HallarMáx (algoritmo 1.3) con $n = 4$. El argumento de árbol de decisión no da una buena cota inferior porque no es fácil determinar cuántas hojas contendrán repeticiones de un resultado específico.

En vez de un árbol de decisión, emplearemos una técnica llamada *argumentos de adversario* para establecer mejores cotas inferiores para el problema de selección. Describiremos esa técnica a continuación.

5.1.3 Argumentos de adversario

Supóngase que estamos jugando un juego de adivinar con un amigo. Escogemos una fecha (mes y día) y el amigo tratará de adivinar la fecha haciendo preguntas que se contestan sí o no. Queremos obligar a nuestro amigo a hacer el mayor número posible de preguntas. Si la primera pregunta es: "¿Es en invierno?" y somos un buen adversario, contestaremos "No" porque hay más fechas en las otras tres estaciones. A la pregunta, "¿La primera letra del nombre del mes está en la primera mitad del alfabeto?" deberemos contestar "Sí". ¿Estamos haciendo trampa?, ¡No hemos escogido realmente una fecha!, de hecho, no escogeremos un mes ni un día específico hasta que la necesidad de mantener la consistencia de nuestras respuestas no nos deje más alternativa. Tal vez ésta no sea una forma muy amistosa de jugar a las adivinanzas, pero es justo la correcta para encontrar cotas inferiores del comportamiento de un algoritmo.

Supóngase que tenemos un algoritmo que creemos es eficiente. Imaginemos un adversario que quiere demostrar lo contrario. En cada punto del algoritmo en el que se toma una decisión (una comparación de claves, por ejemplo), el adversario nos dice el resultado de la decisión. El adversario escoge sus respuestas tratando de obligar al algoritmo a trabajar lo más posible, es decir, a tomar muchas decisiones. Podríamos pensar que el adversario está construyendo gradualmente una entrada "mala" para el algoritmo al tiempo que contesta las preguntas. La única restricción sobre las respuestas del adversario es que deben ser congruentes internamente; debe existir *alguna* entrada del problema para la que las respuestas del adversario fueran correctas. Si el adversario puede obligar al algoritmo a ejecutar $f(n)$ pasos, entonces $f(n)$ será una cota inferior del número de pasos que se ejecutan en el peor caso. Este enfoque se explora en el ejercicio 5.2 para ordenar y fusionar comparando claves.

De hecho, "diseñar contra un adversario" es con frecuencia una buena técnica para resolver de manera eficiente un problema basado en comparaciones. Al pensar en qué comparación hacer en una situación dada, imaginamos que el adversario dará la respuesta menos favorable, entonces escogemos una comparación en la que ambos resultados son aproximadamente tan favorables el uno como el otro. Esta técnica se analiza con mayor detalle en la sección 5.6. Aquí, empero, lo que nos interesa primordialmente es el papel de los argumentos de adversario en los argumentos de cota inferior.

Queremos encontrar una cota inferior para la complejidad de un *problema*, no sólo de un algoritmo específico. Cuando usemos argumentos de adversario, supondremos que el algoritmo es cualquier algoritmo de la clase que se está estudiando, como hicimos con los argumentos de árbol de decisión. Para obtener una buena cota inferior, necesitaremos crear un adversario astuto que pueda frustrar a cualquier algoritmo.

5.1.4 Torneos

En el resto de este capítulo presentaremos algoritmos para problemas de selección y argumentos de adversario de cota inferior para varios casos, incluida la mediana. En la mayor parte de los algoritmos y argumentos usaremos la terminología de los concursos (o torneos) para describir los resultados de las comparaciones. Llamaremos *ganador* al comparando que resulte ser mayor; el otro será el *perdedor*.

5.2 Determinación de `max` y `min`

En toda esta sección usaremos los nombres `max` y `min` para referirnos a las claves más grande y más pequeña, respectivamente, de un conjunto de n claves.

Podemos hallar a `max` y `min` utilizando el algoritmo 1.3 para determinar `max`, eliminando a `max` del conjunto y usando después la variante apropiada del algoritmo 1.3 para encontrar a `min` entre las $n - 1$ claves restantes. Así, podemos hallar a `max` y `min` efectuando $(n - 1) + (n - 2)$ o $2n - 3$ comparaciones. Esto no es óptimo. Aunque sabemos (por el capítulo 1) que se necesitan $n - 1$ comparaciones para encontrar a `max` o `min` de forma independiente, cuando se buscan ambas es posible "compartir" una parte del trabajo. En el ejercicio 1.25 se pedía un algoritmo para hallar a `max` y `min` con únicamente $3n/2$ comparaciones de claves, aproximadamente. Una solución (con n par) consiste en aparear las claves y efectuar $n/2$ comparaciones, luego hallar el mayor de los ganadores y, por separado, el menor de los perdedores. Si n es impar, podría ser necesario considerar a la última clave entre los ganadores y los perdedores. En cualquier caso, el número total de comparaciones es $\lceil 3n/2 \rceil - 2$. En esta sección presentaremos un argumento de adversario para demostrar que esta solución es óptima. Específicamente, en el resto de esta sección demostraremos:

Teorema 5.1 Cualquier algoritmo para hallar `max` y `min` de n claves por comparación de claves deberá efectuar por lo menos $3n/2 - 2$ comparaciones de claves en el peor caso.

Demostración Para establecer la cota inferior podemos suponer que las claves son distintas. Para saber que una clave x es `max` y una clave y es `min`, un algoritmo debe saber que todas las otras claves aparte de x han perdido alguna comparación y que todas las demás claves aparte de y han ganado alguna comparación. Si contamos cada victoria y cada derrota como una unidad de información, un algoritmo deberá contar con (al menos) $2n - 2$ unidades de información para tener la

Situación de las claves x y y comparadas por un algoritmo	Respuesta del adversario	Nueva situación	Unidades de información nueva
N, N	$x > y$	G, P	2
G, N o $G\ P, N$	$x > y$	G, P o $G\ P, P$	1
P, N	$x < y$	P, G	1
G, G	$x > y$	$G, G\ P$	1
P, P	$x > y$	$G\ P, P$	1
G, P o $G\ P, P$ o $G, G\ P$	$x > y$	Sin cambio	0
$G\ P, G\ P$	Congruente con los valores asignados	Sin cambio	0

Tabla 5.1 Estrategia del adversario para el problema de `max` y `min`

certeza de dar la respuesta correcta. Presentamos una estrategia que un adversario puede usar para responder a las comparaciones a modo de revelar el menor número posible de unidades de información nuevas con cada comparación. Imaginemos que el adversario construye un conjunto de entrada específico conforme responde a las comapraciones del algoritmo.

Denotamos la situación de cada clave en cualquier momento durante el curso del algoritmo como sigue:

Situación de la clave	Significado
G	Ganó al menos una comparación y nunca ha perdido
L	Perdió al menos una comparación y nunca ha ganado
$G\ P$	Ha ganado y ha perdido al menos una comparación
N	Todavía no ha participado en una comparación

Cada G o P es una unidad de información. Una situación de N no comunica información. La estrategia del adversario se describe en la tabla 5.1. El punto principal es que, con excepción del caso en el que ninguna de las dos claves ha participado aún en alguna comparación, el adversario puede dar una respuesta que proporcione cuando más una unidad de información nueva. Necesitaremos verificar que si el adversario sigue estas reglas, sus respuestas sean congruentes con alguna entrada. Luego necesitaremos demostrar que tal estrategia obliga a cualquier algoritmo a efectuar el número de comparaciones que dice el teorema.

Obsérvese que, en todos los casos de la tabla 5.1 con excepción del último, la clave escogida por el adversario como ganadora todavía no ha perdido en ninguna comparación, o bien la clave escogida como perdedora todavía no ha ganado. Consideremos la primera posibilidad; supóngase que el algoritmo compara x y y, el adversario escoge x como ganadora, y x no ha perdido aún en ninguna comparación. Incluso si el valor que el adversario había asignado antes a x es menor que el valor que asignó a y, el adversario puede cambiar el valor de x para que supere a y sin contradecir ninguna de las respuestas que dio antes. La otra situación, donde la clave que será la perdedora nunca ha ganado antes, se puede manejar de forma similar: reduciendo el valor de la clave

Comparación	x_1		x_2		x_3		x_4		x_5		x_6	
	Sit.	Valor	Sit.	Valor	Sit.	Valor	Sit.	Valor	Sit.	Valor	Sit.	Valor
x_1, x_2	*G*	20	*P*	10	N	*	N	*	N	*	N	*
x_1, x_5	*G*	20							*P*	5		
x_3, x_4					*G*	15	*P*	8				
x_3, x_6					*G*	15					*P*	12
x_3, x_1	*G P*	20			*G*	*25*						
x_2, x_4			*G P*	10			*P*	8				
x_5, x_6									*G P*	5	*P*	3
x_6, x_4							*P*	2			*G P*	3

Tabla 5.2 Ejemplo de la estrategia del adversario para `max` y `min`

si es necesario. Así, el adversario puede construir una entrada congruente con las reglas de la tabla para responder a las comparaciones del algoritmo. Esto se ilustra en el ejemplo que sigue.

Ejemplo 5.1 Construcción de una entrada empleando las reglas del adversario

La primera columna de la tabla 5.2 muestra una sucesión de comparaciones que podría efectuar algún algoritmo. El resto de las columnas muestra la situación y el valor asignados a las claves por el adversario. (Las claves a las que todavía no se ha asignado un valor se denotan con asteriscos.) Las filas después de la primera sólo contienen los elementos que atañen a la comparación en curso. Obsérvese que, cuando se comparan x_3 y x_1 (en la quinta comparación), el adversario aumenta el valor de x_3 porque se supone que x_3 debe ganar. Más adelante, el adversario cambia los valores de x_4 y x_6 en congruencia con sus reglas. Después de las primeras cinco comparaciones, todas las claves con excepción de x_3 han perdido al menos una vez, así que x_3 es `max`. Después de la última comparación, x_4 es la única clave que nunca ha ganado, así que es `min`. En este ejemplo, el algoritmo realizó ocho comparaciones; la cota de peor caso para seis claves (que falta demostrar) es $3/2 \times 6 - 2 = 7$. ■

Para completar la demostración del teorema 5.1, sólo tenemos que demostrar que las reglas del adversario obligarán a cualquier algoritmo a efectuar al menos $3n/2 - 2$ comparaciones para obtener las $2n - 2$ unidades de información que necesita. El único caso en el que un algoritmo puede obtener dos unidades de información de una comparación es aquel en el que ninguna de las dos claves se había incluido antes en alguna comparación anterior. Supóngase por el momento que n es par. Un algoritmo puede efectuar cuando más $n/2$ comparaciones de claves que no había visto antes, de modo que puede obtener cuando más n unidades de información de esta manera. De todas las demás comparaciones, obtendrá cuando más una unidad de información. El algoritmo necesita otras $n - 2$ unidades de información, por lo que deberá efectuar por lo menos $n - 2$ comparaciones más. Así pues, para obtener $2n - 2$ unidades de información, el algoritmo tendrá que hacer por lo menos $n/2 + n - 2 = 3n/2 - 2$ comparaciones en total. El lector puede verificar fácilmente que, si n es impar, se necesitan al menos $3n/2 - 3/2$ comparaciones. Esto completa la demostración del teorema 5.1. □

5.3 Cómo hallar la segunda llave más grande

Podemos encontrar el segundo elemento más grande de un conjunto hallando y eliminando el más grande, y encontrando después el elemento más grande de los que quedan. ¿Existe un método más eficiente? ¿Podemos demostrar que un método dado es óptimo? En esta sección contestaremos estas preguntas.

5.3.1 Introducción

En toda esta sección usaremos `max` y `segundoMayor` para referirnos a las claves más grande y segunda más grande, respectivamente. A fin de simplificar la descripción del problema y los algoritmos, supondremos que las claves son distintas.

La segunda clave más grande se puede hallar con $2n - 3$ comparaciones utilizando HallarMáx (algoritmo 1.3) dos veces, pero es poco probable que esto sea óptimo. Cabe esperar que parte de la información que el algoritmo descubrió al buscar `max` pueda servir para reducir el número de comparaciones hechas al buscar `segundoMayor`. Específicamente, cualquier clave que pierda con cualquier otra clave distinta de `max` no podrá ser `segundoMayor`. Todas las claves de ese tipo que se descubran durante la búsqueda de `max` se podrán pasar por alto durante la segunda pasada por el conjunto. (El problema de saber cuáles son se considerará más adelante.)

Si aplicamos el algoritmo 1.3 a un conjunto de cinco claves, los resultados podrían ser los siguientes:

Comparandos	Ganador
x_1, x_2	x_1
x_1, x_3	x_1
x_1, x_4	x_4
x_4, x_5	x_4

Entonces, `max` $= x_4$ y `segundoMayor` es x_5 o bien x_1, porque tanto x_2 como x_3 perdieron con x_1. Por tanto, sólo se necesita una comparación más para hallar `segundoMayor` en este ejemplo.

Sin embargo, podría suceder que durante la primera pasada por el conjunto en busca de `max` no obtengamos información útil para determinar `segundoMayor`. Si `max` fuera x_1, entonces todas las demás claves se compararían únicamente con `max`. ¿Significa esto que en el peor caso es preciso efectuar $2n - 3$ comparaciones para hallar `segundoMayor`?, no necesariamente. En el análisis anterior usamos un algoritmo específico, el algoritmo 1.3. Ningún algoritmo puede hallar `max` efectuando menos de $n - 1$ comparaciones, pero otro algoritmo podría proporcionar más información que sirva para eliminar algunas claves de la segunda pasada por el conjunto. El método de torneo, que describiremos a continuación, proporciona tal información.

5.3.2 El método de torneo

El método de torneo tiene ese nombre porque efectúa comparaciones de la misma manera en que se celebran torneos. Las claves se aparean y se comparan en "rondas". En cada ronda después de la primera, los ganadores de la ronda anterior se aparean y comparan. (Si en cualquier ronda el número de claves es impar, una de ellas simplemente espera la siguiente ronda.) Un torneo puede describirse con un diagrama de árbol binario como el que se muestra en la figura 5.1. Cada hoja contiene una clave y en cada nivel subsiguiente el padre de cada par contiene al ganador. La raíz contiene la clave más grande.

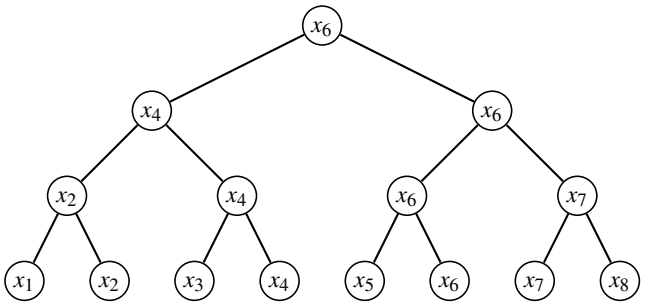


Figura 5.1 Ejemplo de torneo; `max` = x_6; `segundoMayor` podría ser x_4, x_5 o x_7

Al igual que en el algoritmo 1.3, se efectúan $n - 1$ comparaciones para hallar `max`.

Durante la búsqueda de `max`, todas las claves menos `max` pierden en una comparación. ¿Cuántas pierden directamente con `max`? Si n es una potencia de 2, habrá exactamente lg n rondas; en general, el número de rondas es $\lceil \lg n \rceil$. Puesto que `max` interviene en cuando más una comparación en cada ronda, habrá cuando más $\lceil \lg n \rceil$ claves que sólo hayan perdido con `max`. Éstas son las únicas claves que podrían ser `segundoMayor`. Podemos usar el método del algoritmo 1.3 para encontrar la mayor de estas $\lceil \lg n \rceil$ claves efectuando cuando más $\lceil \lg n \rceil - 1$ comparaciones. Así, el torneo encuentra `max` y `segundoMayor` realizando un total de $n + \lceil \lg n \rceil - 2$ comparaciones en el peor caso. Esto es mejor que nuestro primer resultado de $2n - 3$. ¿Podemos mejorar?

5.3.3 Un argumento de adversario de cota inferior

Los dos métodos que consideramos para encontrar la segunda clave más grande buscaron primero la clave más grande. Esta labor no es un desperdicio; cualquier algoritmo que encuentre `segundoMayor` deberá encontrar también `max` porque, para saber que una clave es la segunda más grande, es preciso saber que no es la más grande; es decir, deberá haber perdido en una comparación. Desde luego, la ganadora de la comparación en la que `segundoMayor` pierde debe ser `max`. Este argumento proporciona una cota inferior para el número de comparaciones que necesitamos efectuar para hallar `segundoMayor`, a saber, $n - 1$, porque ya sabemos que se necesitan $n - 1$ comparaciones para hallar `max`. No obstante, cabría esperar que esta cota inferior pueda mejorarse porque un algoritmo para hallar `segundoMayor` deberá tener que efectuar más trabajo que un algoritmo para hallar `max`. Demostraremos el teorema siguiente, que tiene como corolario que el método de torneo es óptimo.

Teorema 5.2 Cualquier algoritmo (que opere comparando claves) para hallar el segundo elemento más grande de un conjunto deberá efectuar por lo menos $n + \lceil \lg n \rceil - 2$ comparaciones en el peor caso.

Demostración En el peor caso, podemos suponer que todas las claves son distintas. Ya observamos que debe haber $n - 1$ comparaciones con perdedoras distintas. Si `max` fue uno de los comparandos en $\lceil \lg n \rceil$ de esas comparaciones, todas menos una de las $\lceil \lg n \rceil$ claves que perdieron con `max` deberán perder otra vez para poder determinar correctamente `segundoMayor`. Entonces, se haría un total de por lo menos $n + \lceil \lg n \rceil - 2$ comparaciones. Por tanto, demostraremos que exis-

te una estrategia de adversario que puede obligar a cualquier algoritmo que busque `segundoMayor` a comparar `max` con $\lceil \lg n \rceil$ claves distintas.

El adversario asigna un "peso" $w(x)$ a cada clave x del conjunto. En un principio, $w(x) = 1$ para todas las x. Cuando el algoritmo compara dos claves x y y, el adversario determina su respuesta y modifica los pesos como sigue.

Caso	Respuesta del adversario	Actualización de pesos
$w(x) > w(y)$	$x > y$	Nuevo $w(x)$ = previo $(w(x) + w(y))$; nuevo $w(y) = 0$.
$w(x) = w(y) > 0$	Igual que arriba.	Igual que arriba.
$w(y) > w(x)$	$y > x$	Nuevo $w(y)$ = previo $(w(x) + w(y))$; nuevo $w(x) = 0$.
$w(x) = w(y) = 0$	Congruente con respuestas previas.	Sin cambio.

Para interpretar los pesos y las reglas del adversario, imaginemos que el adversario construye árboles para representar las relaciones de orden entre las claves. Si x es el padre de y, entonces x venció a y en una comparación. La figura 5.2 muestra un ejemplo. El adversario sólo combina dos árboles cuando se comparan sus raíces. Si el algoritmo compara nodos que no son raíces, no se modifican los árboles. El peso de una clave es simplemente el número de nodos que hay en el árbol de esa clave, si es una raíz, y cero si no es una raíz.

Necesitamos verificar que el adversario siga esta estrategia, que sus respuestas sean congruentes con alguna entrada, y que `max` se compare con por lo menos $\lceil \lg n \rceil$ claves distintas. Estas conclusiones se siguen de una serie de observaciones sencillas:

1. Una clave ha perdido una comparación si y sólo si su peso es cero.
2. En los primeros tres casos, la clave escogida como ganadora tiene peso distinto de cero, así que todavía no ha perdido. El adversario puede darle un valor arbitrariamente grande para asegurar que gane sin contradecir ninguna de sus respuestas anteriores.
3. La sumatoria de los pesos siempre es n. Esto se cumple inicialmente, y la sumatoria se conserva durante la actualización de los pesos.
4. Cuando el algoritmo para, sólo una clave puede tener peso distinto de cero. De lo contrario, habría al menos dos claves que nunca perdieron en una comparación y el adversario podría escoger valores que hicieran incorrecta la decisión del algoritmo respecto a cuál clave es `segundoMayor`.

Sea x la clave cuyo peso no es cero cuando el algoritmo para. Por las observaciones 1 y 4, $x =$ `max`. Por la observación 3, $w(x) = n$ cuando el algoritmo para.

Para completar la demostración del teorema necesitamos mostrar que x ha ganado directamente contra al menos $\lceil \lg n \rceil$ claves distintas. Sea $w_k = w(x)$ inmediatamente después de la k-ésima comparación ganada por x contra una clave hasta entonces invicta. Entonces, por las reglas del adversario,

$$w_k \leq 2w_{k-1}.$$

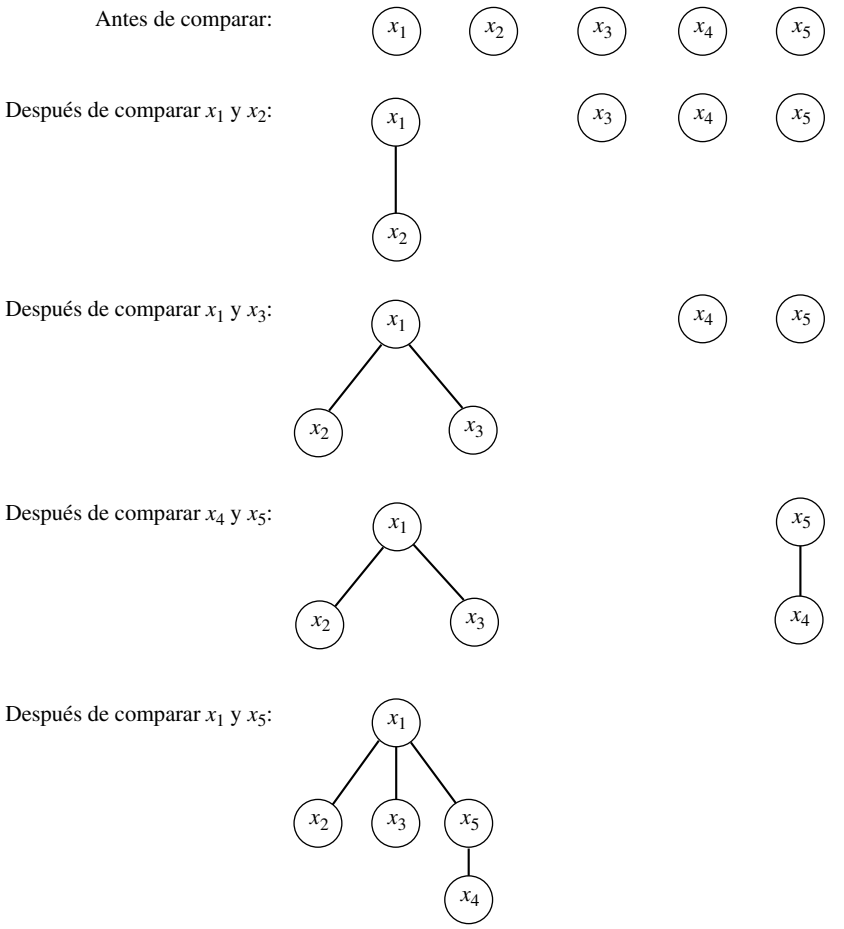


Figura 5.2 Árboles para las decisiones del adversario en el ejemplo 5.2

Ahora sea K el número de comparaciones que x gana contra claves previamente invictas. Entonces

$$n = w_K \geq 2^K \, w_0 = 2^K.$$

Para $K \geq \lg n$, puesto que K es un entero, entonces $K \geq \lceil \lg n \rceil$. Las K claves que contamos aquí son, por supuesto, distintas, ya que una vez derrotada por x, una clave pierde la calidad de invicta y no se le volverá a contar (aunque un algoritmo cometa la torpeza de volver a compararla con x). □

Ejemplo 5.2 La estrategia del adversario en acción

A fin de ilustrar lo que hace el adversario y mostrar cómo sus decisiones corresponden a la construcción paso por paso de una entrada, presentaremos un ejemplo con $n = 5$. Las claves del conjunto que todavía no se han especificado se denotan con asteriscos. Así, en un principio las claves

Comparandos	Pesos	Ganador	Nuevos pesos	Claves
x_1, x_2	$w(x_1) = w(x_2)$	x_1	2, 0, 1, 1, 1	20, 10, *, *, *
x_1, x_3	$w(x_1)$, $w(x_3)$	x_1	3, 0, 0, 1, 1	20, 10, 15, *, *
x_5, x_4	$w(x_5) = w(x_4)$	x_5	3, 0, 0, 0, 2	20, 10, 15, 30, 40
x_1, x_5	$w(x_1)$, $w(x_5)$	x_1	5, 0, 0, 0, 0	41, 10, 15, 30, 40

Tabla 5.3 Ejemplo de la estrategia del adversario para el problema de la Segunda Clave Más Grande

son *, *, *, *, *. Cabe señalar que los valores asignados a algunas claves podrían modificarse posteriormente. Véase la tabla 5.3, que muestra las primeras comparaciones (las que hallan `max`, pero que no bastan para hallar `segundoMayor`). Ninguna comparación subsiguiente modificará los pesos ni los valores asignados a las claves. ■

5.3.4 Implementación del método de torneo para hallar `max` y `segundoMayor`

Para celebrar el torneo que halla `max` necesitamos seguir la pista de alguna manera a los ganadores de cada ronda. Una vez que el torneo haya encontrado a `max`, sólo será necesario comparar las claves que hayan perdido con `max` para encontrar a `segundoMayor`. ¿Cómo podemos saber cuáles elementos pierden con `max` si no sabemos con antelación cuál clave es `max`? Puesto que el torneo es conceptualmente un árbol binario lo más equilibrado posible, podríamos usar la *estructura de montón* de la sección 4.8.1. Con un conjunto de n elementos, usamos una estructura de montón con $2n - 1$ nodos; es decir, un arreglo `E[1]`, ..., `E[2*n-1]`. En un principio, colocamos los elementos en las posiciones $n, \ldots, 2n - 1$. Conforme avanza el torneo, las posiciones $1, \ldots, n - 1$ se llenarán (en orden inverso) con ganadores. El ejercicio 5.4 cubre los demás pormenores. Este algoritmo ocupa un espacio extra lineal y se ejecuta en tiempo lineal.

5.4 El problema de selección

Supóngase que nos interesa hallar la mediana de n elementos que están en un arreglo E en las posiciones $1, \ldots, n$. (Es decir, queremos el elemento de rango $\lceil n/2 \rceil$.) En secciones anteriores descubrimos métodos eficientes para hallar rangos cercanos a un extremo o al otro, como el máximo, el mínimo, tanto el máximo como el mínimo, y la segunda clave más grande. Los ejercicios exploran más variaciones, pero todas las técnicas para resolver estos problemas pierden eficiencia a medida que nos alejamos de los extremos, por tanto no son útiles para hallar la mediana. Si queremos encontrar una solución que sea más eficiente que el simple ordenamiento de todo el conjunto, necesitamos una idea nueva.

5.4.1 Un enfoque de Divide y vencerás

Supóngase que podemos dividir las claves en dos conjuntos S_1 y S_2, tales que todas las claves de S_1 sean menores que todas las claves de S_2. Entonces la mediana estará en el mayor de los dos conjuntos (es decir, el conjunto que tiene más claves, no necesariamente el conjunto que tiene claves más grandes). Podemos hacer caso omiso del otro conjunto y restringir nuestra búsqueda al conjunto más grande.

Pero, ¿qué clave buscamos en el conjunto mayor?, su mediana no es la mediana del conjunto original de claves.

Ejemplo 5.3 Partir para buscar la mediana

Supóngase que $n = 255$. Estamos buscando la mediana de los elementos (el elemento con rango $k = 128$). Supóngase que, después de partir el conjunto, S_1 tiene 96 elementos y S_2 tiene 159. Entonces la mediana de todo el conjunto estará en S_2 y será el 32o. elemento más pequeño de S_2. Por tanto, el problema se reduce a encontrar el elemento con rango 32 de S_2, que tiene 159 elementos. ■

El ejemplo muestra que este enfoque para resolver el problema de la mediana sugiere de manera natural que resolvamos el problema de selección general.

Así pues, estamos desarrollando una solución tipo dividir y vencer para el problema de selección general que, al igual que Búsqueda Binaria y Reparar Montón, divide el problema a resolver en *dos* problemas más pequeños, pero sólo tiene que resolver *uno* de esos problemas más pequeños. Quicksort usa Partir para dividir los elementos en subintervalos de elementos "pequeños" y "grandes" en comparación con un elemento pivote (véase el algoritmo 4.2). Podemos usar una versión modificada de Quicksort para el problema de selección, llamado `hallarKesimo`, en el que sólo es preciso resolver un subproblema recursivo. Los detalles se precisan en el ejercicio 5.8.

En las partes de análisis del ejercicio 5.8 descubrimos el mismo patrón que surgió cuando analizamos Quicksort. Aunque `hallarKesimo` funciona bien en promedio, el peor caso adolece del mismo problema que Quicksort: el elemento pivote podría dar una división muy dispareja de los elementos en S_1 y S_2. Para desarrollar una mejor solución, consideremos lo que aprendimos con Quicksort.

En vista de que el meollo del problema consiste en escoger un "buen" elemento pivote, podemos repasar las sugerencias de la sección 4.4.4, pero ninguna de ellas garantiza que el pivote dividirá el conjunto de elementos en subconjuntos del mismo, o casi del mismo tamaño. En la sección que sigue veremos que, si hacemos un esfuerzo un poco mayor, es posible escoger un pivote cuya "bondad" está garantizada. Sabremos con certeza que cada conjunto tendrá por lo menos $0.3n$ y cuando más $0.7n$ elementos. Con este elemento pivote de "alta calidad", el método de dividir y vencer funciona de manera eficiente en el peor caso y también en el caso promedio.

⋆ 5.4.2 Un algoritmo de selección en tiempo lineal

El algoritmo que presentamos en esta sección es una simplificación del primer algoritmo lineal descubierto para resolver el problema de selección. La simplificación hace más comprensible la estrategia general (aunque los detalles son complicados y se requiere ingenio para implementarlos), pero es menos eficiente que el original. El algoritmo es importante e interesante porque resuelve el problema de selección en general, no sólo el de la mediana, porque *es* lineal y porque hizo posible desarrollar mejoras.

Como ha sido nuestra costumbre, simplificaremos la descripción del algoritmo suponiendo que todas las claves son distintas. No es difícil modificarlo para el caso en que hay claves repetidas.

Algoritmo 5.1 Selección

Entradas: S, un conjunto de n claves; y k, un entero tal que $1 \leq k \leq n$.

Salidas: La k-ésima clave más pequeña de S.

Comentario: Recordemos que $|S|$ denota el número de elementos en S.

```
Elemento seleccionar(ConjuntoDeElementos S, int k)
0. if (|S| ≤ 5)
     return solución directa para el k-ésimo elemento de S.
```

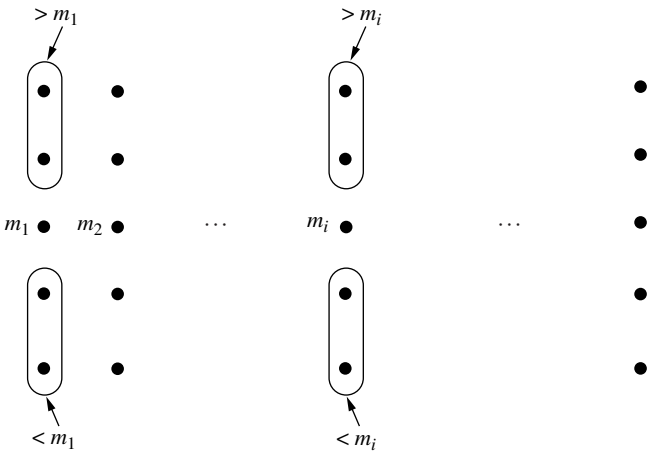
1. Dividir las claves en conjuntos de cinco cada uno y hallar la mediana de cada conjunto. (El último conjunto podría tener menos de cinco claves; no obstante, las referencias posteriores a "conjunto de cinco claves" también incluyen a este conjunto.) Llamamos M al conjunto de medianas. Sea $n_M = |M| = \lceil n/5 \rceil$. En este punto podemos imaginar que las claves están dispuestas como se muestra en la figura 5.3(a). En cada conjunto de cinco claves, las dos mayores que la mediana aparecen arriba de la mediana y las dos menores aparecen abajo de la mediana.

2. m^* = `seleccionar`$(M, \lceil |M|/2 \rceil)$;
   (Ahora m^* es la mediana de M, es decir, la mediana de las medianas.)
   Imaginemos ahora las claves como están en la figura 5.3(b), donde se han reacomodado los conjuntos de cinco claves de modo que los conjuntos cuyas medianas son mayores que m^* aparecen a la derecha del conjunto que contiene a m^* y los conjuntos con medianas menores aparecen a la izquierda del conjunto que contiene a m^*. Observemos que, por transitividad, todas las claves de la sección rotulada B son mayores que m^* y todas las claves de la sección rotulada C son menores que m^*.

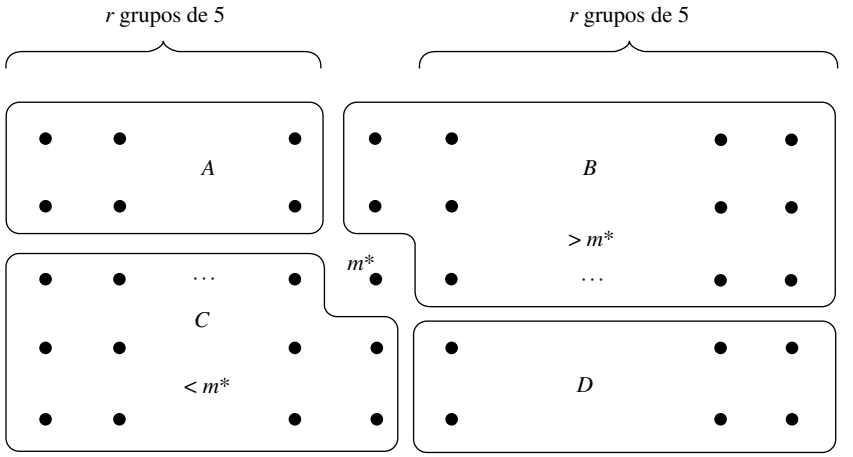
3. Comparamos con m^* cada una de las claves de las secciones rotuladas A y D en la figura 5.3(b).
   Sea $S_1 = C \cup \{\text{claves de } A \cup D \text{ que son menores que } m^*\}$.
   Sea $S_2 = B \cup \{\text{claves de } A \cup D \text{ que son mayores que } m^*\}$.
   Aquí termina el proceso de partir, m^* es el pivote.

4. Dividir y vencer:
   if $(k = |S_1| + 1)$
      m^* es la k-ésima clave más pequeña, así que:
      return m^*;
   else if $(k \leq |S_1|)$
      la k-ésima clave más pequeña está en S_1, así que:
      return `seleccionar`(S_1, k);
   else
      la k-ésima clave más pequeña está en S_2, así que:
      return `seleccionar`$(S_2, k - |S_1| - 1)$;

El algoritmo 5.1 se expresa en términos de un conjunto S y un rango k. Aquí analizaremos brevemente la implementación con un arreglo E, empleando los índices 1 a n, en lugar de 0 a $n - 1$. Hallar un elemento con rango k equivale a contestar la pregunta: si este arreglo estuviera ordenado, ¿cuál elemento estaría en `E[k]`? Si S_1 tiene n_1 elementos, entonces reacomodaremos E



(a) m_i es la mediana del i-ésimo grupo de cinco claves. Hay $n_M = \lceil n/5 \rceil$ grupos. Por sencillez, en este diagrama se supone que n es un múltiplo de 5.



(b) Las medianas menores que m^* están a su izquierda; las medianas mayores que m^* están a su derecha. Por sencillez, este diagrama supone que n es un múltiplo impar de 5, así que $n = 5(2r + 1)$ para alguna r.

Figura 5.3 Pasos 1 y 2 del algoritmo de selección en tiempo lineal

de modo que todos los elementos de S_1 estén en las posiciones $1, \ldots, n_1$, m^* esté en la posición $n_1 + 1$ y todos los elementos de S_2 estén en las posiciones $n_1 + 2, \ldots, n$.

Primero observamos que si $k = n_1 + 1$, m^* es el elemento deseado. Si $k \leq n_1$, entonces la pregunta para la siguiente invocación de `seleccionar` será: si el segmento `E[1]`, ..., `E[`n_1`]` estuviera ordenado, ¿cuál elemento estaría en `E[k]`? Si $k \geq n_1 + 2$, la pregunta para la siguiente in-

vocación de `seleccionar` será: si el segmento `E[`$n_1 + 2$`],...,` `E[n]` estuviera ordenado, ¿cuál elemento estaría en `E[k]`?, (esto equivale al problema de hallar un elemento de rango $k - n_1 - 1$ en el conjunto S_2 solo). Lo importante aquí es que la variable k será la misma para todas las invocaciones recursivas. Sin embargo, es preciso hacer ciertas modificaciones a los detalles de las pruebas para determinar en cuál subintervalo se hará la búsqueda recursiva. Esto se deja como ejercicio (ejercicio 5.9).

⋆ 5.4.3 Análisis del algoritmo de selección

A continuación mostraremos que `seleccionar` es un algoritmo lineal. No haremos una demostración completa, pero sí presentaremos la estructura del argumento suponiendo que n es un múltiplo impar de 5 a fin de simplificar el conteo.

Sea $W(n)$ el número de comparaciones de claves que `seleccionar` efectúa en el peor caso con entradas de n claves. Suponiendo $n = 5(2r + 1)$ para algún entero no negativo r (y haciendo caso del problema de que esta condición tal vez no se cumpla para los tamaños de las entradas de las invocaciones recursivas), contaremos las comparaciones efectuadas en cada paso de `seleccionar`. Después de algunos de los pasos daremos explicaciones breves de los cálculos.

1. Hallar las medianas de conjuntos de cinco claves: $6(n/5)$ comparaciones.
   La mediana de cinco claves se puede hallar con seis comparaciones (ejercicio 5.14). Hay $n/5$ conjuntos de cinco claves.
2. Hallar recursivamente la mediana de las medianas: $W(n/5)$ comparaciones.
3. Comparar todas las claves de las secciones A y D con m^* (véase la figura 5.3b): $4r$ comparaciones.
4. Invocar recursivamente a `seleccionar`: $W(7r + 2)$ comparaciones.
   En el peor caso, las $4r$ claves están todas en las secciones A y D y quedarán del mismo lado de m^* (es decir, todas serán menores que m^* o todas serán mayores que m^*). B y C tienen $3r + 2$ elementos cada una, así que el tamaño de la entrada más grande posible para la invocación recursiva de `seleccionar` es $7r + 2$.

Puesto que $n = 5(2r + 1)$, r es aproximadamente $n/10$. Entonces

$$W(n) \leq 1.2n + W(0.2n) + 0.4n + W(0.7n) = 1.6n + W(0.2n) + W(0.7n). \qquad (5.1)$$

Aunque esta ecuación (realmente desigualdad) de recurrencia es del tipo de dividir y vencer, los dos subproblemas no son del mismo tamaño, por lo que no podemos aplicar simplemente el teorema Maestro (teorema 3.17). Sin embargo, podemos desarrollar un árbol de recursión (sección 3.7), como se muestra en la figura 5.4. Puesto que las sumatorias de fila forman una serie geométrica decreciente cuyo cociente es 0.9, el total es Θ del término más grande, que es $\Theta(n)$. La ecuación (1.10) da la expresión exacta para la serie geométrica, que es $16n$ menos un término muy pequeño. Este resultado también se puede verificar por inducción. Por tanto, el algoritmo de selección es un algoritmo lineal.

La presentación original del algoritmo en la literatura incluía mejoras para reducir el número de comparaciones de claves a aproximadamente $5.4\,n$. El mejor algoritmo que se conoce actualmente para hallar la mediana efectúa $2.95\,n$ comparaciones en el peor caso (y también es complicado).

Puesto que `seleccionar` es recursivo, ocupa espacio en una pila; no es un algoritmo en su lugar. Sin embargo, la profundidad de la recursión está en $O(\log n)$, así que es poco probable que esto constituya un problema.

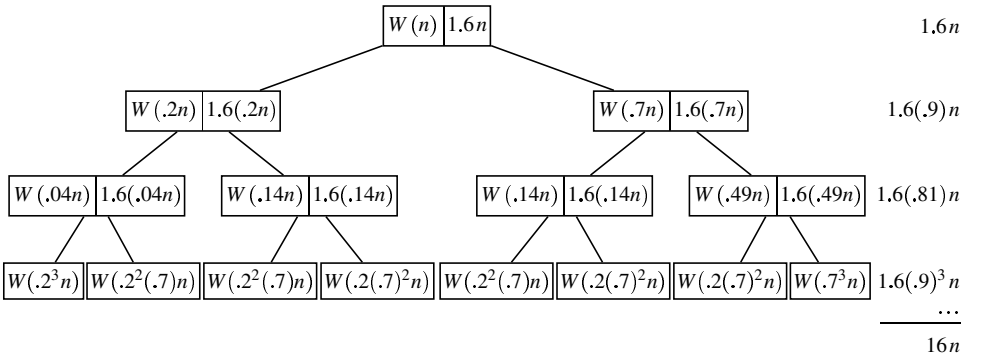


Figura 5.4 Árbol de recursión para `seleccionar`

5.5 Una cota inferior para la determinación de la mediana

Estamos suponiendo que E es un conjunto de n claves y que n es impar. Estableceremos una cota inferior para el número de comparaciones de claves que debe realizar cualquier algoritmo de comparación de claves para encontrar `mediana`, la $[(n + 1)/2]$-ésima clave. Puesto que estamos estableciendo una cota inferior podemos, sin pérdida de generalidad, suponer que las claves son distintas.

Primero afirmamos que, para conocer `mediana`, un algoritmo debe conocer la relación entre cada una de las otras claves y `mediana`. Es decir, para cada una de las demás claves x, el algoritmo debe saber que $x >$ `mediana` o que $x <$ `mediana`. En otras palabras, el algoritmo debe establecer relaciones como se ilustra en el árbol de la figura 5.5. Cada nodo representa una clave y cada rama representa una comparación. La clave que está en el extremo alto de la rama es la más grande. Supóngase que hubiera alguna clave, digamos y, cuya relación con `mediana` se desconociera. (Véase en la figura 5.6(a) un ejemplo.) Un adversario podría modificar el valor de y, pasándola al otro lado de `mediana`, como en la figura 5.6(b), sin contradecir los resultados de ninguna de las comparaciones efectuadas. Entonces `mediana` no sería la mediana; la respuesta del algoritmo sería errónea.

Puesto que hay n nodos en el árbol de la figura 5.5, hay $n - 1$ ramas, así que deben efectuarse por lo menos $n - 1$ comparaciones. Esta cota inferior no es sorprendente ni interesante. Mostraremos que el adversario puede obligar a un algoritmo a efectuar otras comparaciones "inútiles" antes de hacer las $n - 1$ comparaciones que necesita para establecer el árbol de la figura 5.5.

Definición 5.1 Comparación crucial

Una comparación en la que interviene una clave x es una *comparación crucial para x* si es la primera comparación en la que $x > y$, para alguna $y \geq$ `mediana`, o en la que $x < y$ para alguna $y \leq$ `mediana`. Las comparaciones de x y y en las que $x >$ `mediana` y $y <$ `mediana` *no son cruciales*. ■

Una comparación crucial establece la relación entre x y `mediana`. Cabe señalar que la definición no exige que ya se conozca la relación entre y y `mediana` en el momento en que se efectúa la comparación crucial para x.

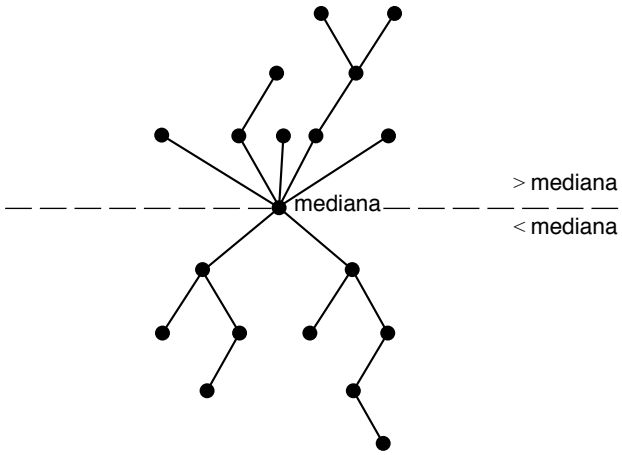
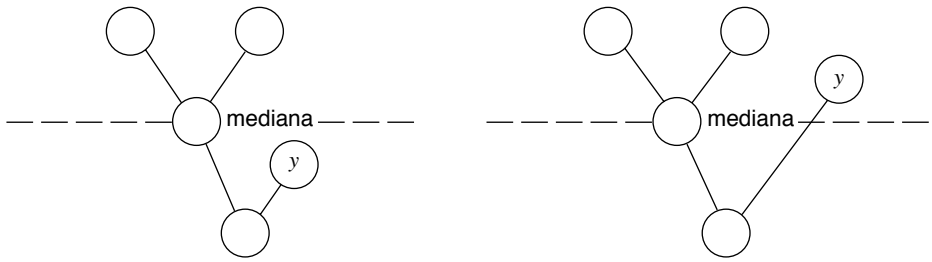


Figura 5.5 Comparaciones que relacionan cada clave con `mediana`



(a) $y <$ mediana. (b) $y >$ mediana; mediana no es la mediana.

Figura 5.6 Un adversario derrota a un mal algoritmo

Presentaremos un adversario que obliga a un algoritmo a efectuar comparaciones *no cruciales*. El adversario escoge algún valor (pero no una clave específica) como `mediana`. Asignará un valor a una clave la primera vez que el algoritmo use esa clave en una comparación. En tanto pueda hacerlo, el adversario asignará valores a claves nuevas implicadas en una comparación tratando de colocar las claves en lados opuestos de `mediana`. El adversario no puede asignar valores mayores que `mediana` a más de $(n - 1)/2$ claves, ni valores menores que `mediana` a más de $(n - 1)/2$ claves. El adversario recuerda las asignaciones que ha hecho, con el fin de no violar esas restricciones. Indicamos la situación de una clave durante la ejecución del algoritmo así:

G	Se le asignó un valor más *G*rande que `mediana`.
P	Se le asignó un valor más *P*equeño que `mediana`.
N	Todavía no ha participado en una comparación.

Comparandos	Acción del adversario
N, N	Hacer una clave mayor que `mediana` y la otra menor.
G, N o *N, G*	Asignar a la clave con situación *N* un valor menor que `mediana`.
P, N o *N, P*	Asignar a la clave con situación *N* un valor mayor que `mediana`.

Tabla 5.4 Estrategia del adversario para el problema de hallar la mediana

La estrategia del adversario se resume en la tabla 5.4. En todos los casos, si ya hay $(n - 1)/2$ claves con situación *P* (o *L*), el adversario hace caso omiso de la regla de la tabla y asigna a las nuevas claves valores más grandes (o más pequeños) que `mediana`. Si sólo queda una clave sin valor, el adversario le asigna el valor `mediana`. Siempre que el algoritmo compara dos claves cuyas situaciones son *G* y *G*, *P* y *P*, o *G* y *P*, el adversario se limita a dar la respuesta correcta con base en los valores que ya asignó a las claves.

Ninguna de las comparaciones descritas en la tabla 5.4 es crucial. ¿Cuántas puede el adversario obligar a cualquier algoritmo a hacer? Cada una de esas comparaciones crea cuando más una clave *G* y cada una crea cuando más una clave *P*. Puesto que el adversario está en libertad de efectuar las asignaciones indicadas hasta que haya $(n - 1)/2$ claves *G* o $(n - 1)/2$ claves *P*, puede obligar a cualquier algoritmo a efectuar por lo menos $(n - 1)/2$ comparaciones no cruciales. (Puesto que un algoritmo podría comenzar haciendo $(n - 1)/2$ comparaciones en las que participan dos claves *N*, este adversario no puede garantizar más de $(n - 1)/2$ comparaciones no cruciales.)

Ahora podemos concluir que el número total de comparaciones debe ser por lo menos $n - 1$ (las comparaciones cruciales) + $(n - 1)/2$ (comparaciones no cruciales). Sintetizamos el resultado en el teorema siguiente.

Teorema 5.3 Cualquier algoritmo para hallar la mediana de n claves (con n impar) por comparación de claves debe efectuar por lo menos $3n/2 - 3/2$ comparaciones en el peor caso. □

Nuestro adversario no fue tan ingenioso como pudo haber sido en su intento por obligar a un algoritmo a realizar comparaciones no cruciales. En los últimos años la cota inferior del problema de la mediana ha ido subiendo lentamente hasta aproximadamente $1.75n - \log n$, luego hasta cerca de $1.8n$, luego un poco más arriba. La mejor cota inferior que se conoce actualmente es un poco mayor que $2n$ (con n grande). Sigue habiendo una pequeña brecha entre la mejor cota inferior conocida y el mejor algoritmo conocido para hallar la mediana.

5.6 Diseño contra un adversario

Diseñar contra un adversario puede ser una herramienta potente para desarrollar un algoritmo con operaciones como las comparaciones, que extraen información acerca de los elementos de la entrada. La idea principal consiste en prever que cualquier "pregunta" (o sea, comparación u otra

prueba realizada por el algoritmo) va a recibir una respuesta escogida por un adversario de modo que sea lo menos favorable que se pueda para el algoritmo, por lo regular dando el mínimo de información posible. Para contrarrestar esto, el algoritmo deberá escoger comparaciones (o las operaciones en cuestión) para las que ambas respuestas proporcionen la misma cantidad de información, en la medida de lo posible.

Ya surgió antes la idea de que un buen algoritmo utiliza algún concepto de equilibrio (cuando estudiamos los árboles de decisión). El número de comparaciones efectuadas en el peor caso es la altura de un árbol de decisión para el algoritmo. Si queremos que la altura no crezca mucho, para un tamaño de problema fijo, deberemos mantener el árbol lo más equilibrado que podamos. Un buen algoritmo escoge comparaciones tales que el número de posibles desenlaces (salidas) para un resultado de la comparación sea aproximadamente igual al número de desenlaces para el otro resultado.

Ya hemos visto varios ejemplos de esta técnica: Mergesort, hallar tanto `max` como `min` y hallar el segundo elemento más grande. La primera fase del método de torneo para hallar el segundo elemento más grande, es decir, el torneo que encuentra el elemento máximo, es el ejemplo más claro. En la primera ronda, cada comparación de claves se efectúa entre dos elementos acerca de los que no se sabe nada, de modo que un adversario no tenga base para favorecer a una respuesta o a la otra. En rondas subsiguientes, en la medida de lo posible, se comparan elementos que han ganado tantas veces como han perdido, para que el adversario nunca pueda dar una respuesta que sea menos informativa que la otra. En contraste, el algoritmo directo para hallar el máximo primero compara x_1 con x_2, luego compara el ganador (digamos x_2) con x_3. En este caso el adversario *sí puede* dar una respuesta que sea menos informativa que la otra. (¿Cuál?)

En general, en problemas basados en comparaciones, la situación completa de un elemento incluye más que el número de victorias y derrotas previas. Más bien, la situación de un elemento incluye el número de elementos que se sabe son menores y el número de elementos que se sabe son mayores por transitividad. Es posible usar estructuras de árbol como las de la figura 5.2 para representar la información de situación gráficamente.

A fin de ilustrar mejor la técnica de diseñar contra un adversario, consideraremos dos problemas cuya solución óptima es difícil: hallar la mediana de cinco elementos y ordenar cinco elementos (ejercicios 5.14 y 5.15). La mediana puede hallarse con seis comparaciones y se pueden ordenar cinco elementos con siete comparaciones. Muchos estudiantes (y profesores) han pasado horas probando diversas estrategias, buscando infructuosamente soluciones. Los algoritmos óptimos extraen la mayor cantidad posible de cada comparación. La técnica que describimos en esta sección es una gran ayuda para "comenzar con el pie derecho". La primera comparación es arbitraria; es por fuerza entre dos claves acerca de las cuales nada sabemos. ¿La segunda comparación debe incluir cualquiera de estas claves?, no; comparar dos claves nuevas, que están en la misma situación, proporciona más información. Ahora tenemos dos claves que (sabemos) son más grandes que otras dos, dos claves que (sabemos) son menores que otras dos y una que no hemos examinado. ¿Cuáles dos compararemos a continuación?

¿El lector está comenzando a preguntarse qué problema estamos tratando de resolver? La técnica de diseñar contra un adversario sugiere las mismas tres primeras comparaciones tanto para el problema de la mediana como para el de ordenar. Terminar los algoritmos tiene sus bemoles y como ejercicio puede enseñarnos mucho.

Ejercicios

Sección 5.1 Introducción

5.1 Dibuje el árbol de decisión para HallarMáx (algoritmo 1.3) con $n = 4$.

5.2 Considere el problema de ordenar n elementos. En esencia, sólo hay $n!$ desenlaces distintos, uno para cada permutación. Los adversarios no están limitados en cuanto a la cantidad de cálculos que pueden efectuar para decidir el desenlace, o respuesta, de una comparación que el algoritmo "pregunta". En principio, un adversario para el problema del ordenamiento podría examinar todas las permutaciones antes de tomar una decisión.

a. Utilice la idea anterior para desarrollar una estrategia de adversario para el ordenamiento basado en comparaciones. Determine una cota inferior con base en su estrategia. Compare su resultado con la cota inferior del teorema 4.10.

★ **b.** Desarrolle una estrategia de adversario para el problema de fusionar dos sucesiones ordenadas, cada una de las cuales contiene $n/2$ claves. Deberá ser una modificación sencilla de su estrategia para la parte (a). Determine una cota inferior para el peor caso de los algoritmos basados en comparaciones que resuelven este problema, con base en su estrategia. Compare su resultado con la cota inferior del teorema 4.4. *Sugerencia:* Estudie el ejercicio 4.25.

Sección 5.2 Determinación de `max` *y* `min`

5.3 Usamos un argumento de adversario para establecer la cota inferior para el problema de hallar la mínima y la máxima de n claves. ¿Qué cota inferior obtenemos con un argumento de árbol de decisión?

Sección 5.3 Cómo hallar la segunda llave más grande

5.4 En este ejercicio usted escribirá un algoritmo basado en la estructura de montón (sección 4.8.1) para hallar `max` y `segundoMayor` empleando el método de torneo.

a. Demuestre que el procedimiento que sigue coloca `max` en `E[1]`. El arreglo E tiene espacio para los índices $1, \ldots, 2n - 1$. (Recuerde que "`ultimo -= 2`" resta 2 a `ultimo`.)

```
hallarMaxEnMonton(Elemento [ ]E, int n)
   int ultimo;
   Colocar n elementos en E[n], ..., E[2*n-1].
   for(ultimo = 2*n-2; ultimo ≥ 2; ultimo -= 2)
      E[ultimo/2] = max(E[ultimo], E[ultimo+1]);
```

b. Explique cómo podemos determinar cuáles elementos perdieron ante el ganador.

c. Complete el código para hallar `segundoMayor` una vez que `hallarMaxEnMonton` termine.

5.5 ¿Cuántas comparaciones hace en promedio el método de torneo para hallar `segundoMayor`

a. si n es una potencia de 2?

b. si n no es una potencia de 2?

Sugerencia: Considere el ejercicio 5.4.

5.6 El algoritmo que sigue halla las claves más grande y segunda más grande de un arreglo E que incluye n claves, examinando de forma secuencial el arreglo y recordando las dos claves más grandes que ha visto hasta el momento. (Supuesto: $n \geq 2$.)

```
if (E[1] > E[2])
   max = E[1];
   segundo = E[2];
else
   max = E[2];
   segundo = E[1];
for (i = 3; i ≤ n; i ++)
   if (E[i] > segundo)
      if (E[i] > max)
        segundo = max;
        max = E[i];
      else
        segundo = E[i];
```

a. ¿Cúantas comparaciones de claves hace este algoritmo en el peor caso? Proporcione una entrada de peor caso para $n = 6$ empleando claves enteras.

★ **b.** ¿Cuántas comparaciones efectúa este algoritmo en promedio con n claves suponiendo que cualquier permutación de las claves (respecto a su orden correcto) es igualmente verosímil?

★ **5.7** Escriba un algoritmo eficiente para hallar la tercera clave más grande entre n claves. ¿Cuántas comparaciones de claves efectúa su algoritmo en el peor caso? ¿Es necesario que semejante algoritmo determine cuál clave es `max` y cuál es `segundoMayor`?

Sección 5.4 El problema de selección

5.8 Es posible modificar Quicksort para que halle la k-ésima clave más pequeña entre n claves de modo que en la mayor parte de los casos efectúe mucho menos trabajo que el que se necesita para ordenar totalmente el conjunto.

a. Escriba un algoritmo Quicksort modificado llamado `hallarKesimo` para realizar esa tarea.

b. Demuestre que si este algoritmo se usa para hallar la mediana, el peor caso está en $\Theta(n^2)$.

c. Plantee una ecuación de recurrencia para el tiempo de ejecución promedio de este algoritmo.

★ **d.** Analice el tiempo de ejecución promedio de su algoritmo. ¿Qué orden asintótico tiene?

5.9 Siguiendo el bosquejo de pseudocódigo del algoritmo de Selección (algoritmo 5.1), analizamos brevemente su implementación con un arreglo. Hallar un elemento con rango k en un arreglo E de n elementos que equivale a contestar la pregunta: Si este arreglo estuviera ordenado, ¿cuál elemento estaría en `E[k]`? El punto era que el parámetro k es el mismo para todas las invocacio-

nes recursivas. Reescriba las condiciones de prueba de los dos enunciados **if** del paso 4 de modo que funcionen correctamente con esta implementación.

5.10 Suponga que usa el algoritmo siguiente para hallar las k claves más grandes en un conjunto de n claves. (Vea los algoritmos de montón en la sección 4.8.)

```
Construir un montón H con las n claves;
for (i = 1; i ≤ k; i ++)
    salida(obtMax(H));
    borrarMax(H);
```

¿Qué tan grande puede ser k (en función de n) para que este algoritmo sea lineal en n?

★ **5.11** Generalice el método de torneo para hallar las k más grandes de n claves (donde $1 \leq k \leq n$). Precise cualesquier detalles de implementación que afecten el orden del tiempo de ejecución. ¿Cuál es la rapidez de su algoritmo en función de n y k?

Sección 5.5 Una cota inferior para la determinación de la mediana

5.12 Suponga que n es par y que definimos la mediana como la $[n/2]$-ésima clave más pequeña. Haga las modificaciones necesarias al argumento de cota inferior y al teorema 5.3 (donde supusimos que n era impar).

Sección 5.6 Diseño contra un adversario

5.13 Los algoritmos que siguen, ¿qué tan bien satisfacen el criterio de efectuar comparaciones en las que ambos desenlaces proporcionen aproximadamente la misma cantidad de información? ¿Cómo respondería un adversario a las comparaciones empleando una estrategia de "mínimo de información nueva"? ¿Hace esto que los algoritmos trabajen con sus peores casos?

a. Ordenamiento por Inserción.
b. Quicksort.
c. Mergesort.
d. Heapsort.
e. Heapsort Acelerado.

★ **5.14** Sugiera un algoritmo para hallar la mediana de cinco claves con sólo seis comparaciones en el peor caso. Describa los pasos, pero no escriba código. Emplear diagramas de árbol como los de la figura 5.2, los cuales podrían ayudar a explicar lo que el algoritmo hace. *Sugerencia:* En la sección 5.6 se bosquejaron parcialmente una estrategia útil y los primeros pasos.

★ **5.15** Sugiera un algoritmo para ordenar cinco claves con sólo siete comparaciones en el peor caso. Describa los pasos, pero no escriba código. El empleo de diagramas de árbol similares a los de la figura 5.2 podria ayudar a explicar lo que hace el algoritmo. *Sugerencia:* En la sección 5.6 se bosquejaron parcialmente una estrategia útil y los primeros pasos.

Problemas adicionales

5.16 Demuestre el teorema 1.16 (la cota inferior para búsquedas en un arreglo ordenado) empleando un argumento de adversario. *Sugerencia:* Defina un *intervalo activo* que consiste en los índices mínimo y máximo del arreglo que podrían contener K, la clave que se está buscando.

5.17 Sea E un arreglo con elementos definidos para los índices $0, \ldots, n$ (o sea que hay $n + 1$ elementos). Suponga que se sabe que E es *unimodal*, lo que significa que `E[i]` crece estrictamente hasta algún índice M y decrece estrictamente para los índices $i > M$. Por tanto, `E[M]` es el máximo. (Cabe señalar que M podría ser 0 o n.) El problema consiste en hallar M.

a. Como "calentamiento", demuestre que, con $n = 2$, se necesitan y bastan dos comparaciones.
b. Escriba un algoritmo para hallar M comparando diversas claves de E.
c. ¿Cuántas comparaciones efectúa su algoritmo en el peor caso? (Deberá poder idear un algoritmo que esté en $o(n)$.)
★ **d.** Suponga que $n = F_k$, el k-ésimo número de Fibonacci, según la definición de la ecuación (1.13), donde $k \geq 2$. Describa un algoritmo para hallar M con $k - 1$ comparaciones. Describa las ideas, pero no escriba código.
★ **e.** Idee una estrategia de adversario que obligue a cualquier algoritmo basado en comparaciones a efectuar por lo menos $\lg n + 2$ comparaciones para hallar M, para $n \geq 4$. Esto demuestra que el problema es por lo menos un poco más difícil que buscar en un arreglo ordenado. *Sugerencia:* Pruebe una versión más compleja de la estrategia de adversario que se sugiere para el ejercicio 5.16.

★ **5.18** Suponga que `E1` y `E2` son arreglos, cada uno con n claves en orden ascendente.

a. Idee un algoritmo $O(\log n)$ para hallar la n-ésima más pequeña de las $2n$ claves. (Se trata de la mediana del conjunto combinado.) Por sencillez, puede suponer que las claves son distintas.
b. Dé una cota inferior para este problema.

5.19

a. Sugiera un algoritmo para determinar si las n claves de un arreglo son todas distintas. Suponga comparaciones de tres vías; es decir, el resultado de una comparación de dos claves es $<$, $=$ o $>$. ¿Cuántas comparaciones de claves efectúa su algoritmo?
★ **b.** Dé una cota inferior para el número de comparaciones de claves (de tres vías) que se necesitan. (Intente que sea $\Omega(n \log n)$.)

5.20 Considere el problema de determinar si una cadena de bits de longitud n contiene dos ceros consecutivos. La operación básica consiste en examinar una posición de la cadena para ver si es un 0 o un 1. Para cada $n = 2, 3, 4, 5$ presente una estrategia de adversario que obligue a cualquier algoritmo a examinar cada uno de los bits, o bien dé un algoritmo que resuelva el problema examinando menos de n bits.

5.21 Suponga que tiene una computadora con memoria pequeña y le dan una sucesión de claves en un archivo externo (en disco o cinta). Se pueden leer claves y colocarlas en la memoria para procesarlas, pero ninguna clave se puede leer más de una vez.

a. ¿Cuántas celdas de memoria (en las que se colocan claves) se necesitan como mínimo para determinar cuál es la clave más grande del archivo? Justifique su respuesta.

b. ¿Cuántas celdas de memoria se necesitan como mínimo para determinar la mediana? Justifique su respuesta.

5.22

a. Le dan n claves y un entero k tal que $1 \leq k \leq n$. Sugiera un algoritmo eficiente para hallar *cualquiera* de las k claves más pequeñas. (Por ejemplo, si $k = 3$, el algoritmo podría devolver la clave más pequeña, la segunda más pequeña o la tercera más pequeña; no necesita conocer el rango exacto de la clave que devuelve.) ¿Cuántas comparaciones de claves efectúa su algoritmo? *Sugerencia:* No busque algo complicado. Basta entender algo para hallar un algoritmo corto y sencillo.

b. Dé una cota inferior, en función de n y k, para el número de comparaciones con que se puede resolver el problema.

★ **5.23** Sea E un arreglo de enteros positivos con n elementos. Un *elemento de mayoría* en E es un elemento que se da *más de* $n/2$ veces en el arreglo. El *problema de elemento de mayoría* consiste en hallar el elemento de mayoría de un arreglo, si existe, o devolver -1 si no existe. Las únicas operaciones que se pueden efectuar con los elementos son compararlos entre sí y cambiarlos de lugar o copiarlos.

Escriba un algoritmo para el problema de elemento de mayoría. Analice el consumo de tiempo y espacio de su algoritmo en el peor caso. (Existen algoritmos $\Theta(n^2)$ fáciles, pero también hay una solución lineal. *Sugerencia:* Utilice una variación de la técnica de la sección 5.3.2.)

★ **5.24** M es una matriz de $n \times n$ enteros en la que las claves de cada fila están en orden creciente (leyendo de izquierda a derecha) y las claves de cada columna están en orden creciente (leyendo de arriba a abajo). Considere el problema de hallar la posición de un entero x en M o determinar que x no está ahí. Dé un argumento de adversario para establecer una cota inferior del número de comparaciones de x con elementos de la matriz que se necesitan para resolver este problema. El algoritmo puede usar comparaciones de tres vías; es decir, una comparación de x con `M[i][j]` dice si $x <$ `M[i][j]`, $x =$ `M[i][j]` o $x >$ `M[i][j]`.

Nota: Hallar un algoritmo eficiente para el problema del ejercicio 4.58 del capítulo 4. Si usted hizo un buen trabajo tanto con su algoritmo como con su argumento de adversario, el número de comparaciones efectuadas por el algoritmo deberá ser igual a su cota inferior.

Notas y referencias

Knuth (1998) es una referencia excelente para el material de este capítulo. Contiene algo de historia del problema de selección, incluido el intento que hizo Charles Dodgson (Lewis Carroll) en 1883 por idear un algoritmo correcto para que el segundo premio en los torneos de tenis se pudiera otorgar de manera justa. El algoritmo de torneo para hallar la segunda clave más grande apare-

ció en un artículo de 1932 escrito por J. Schreier (en polaco). En 1964, S.S. Kislitsin demostró (en ruso) que es óptimo. El argumento de cota inferior que se da aquí se basa en Knuth (1998).

Knuth atribuye a I. Pohl el algoritmo y la cota inferior para hallar `min` y `max`, y el ejercicio 5.21.

El primer algoritmo de selección lineal aparece en Blum, Floyd, Pratt, Rivest y Tarjan (1973). Otros algoritmos de selección y cotas inferiores aparecen en Hyafil (1976), Schönhage, Paterson y Pippenger (1976), y en Dor y Zwick (1995, 1996a, 1996b).

6

Conjuntos dinámicos y búsquedas

6.1 Introducción

Un conjunto dinámico es un conjunto que sufre cambios en cuanto a sus miembros durante los cálculos. En algunas aplicaciones, los conjuntos están inicialmente vacíos y se insertan elementos conforme avanzan los cálculos. Es común que no se conozca con mucha precisión el tamaño máximo que puede alcanzar un conjunto, antes de efectuar los cálculos. Otras aplicaciones parten de un conjunto grande y eliminan elementos conforme avanzan los cálculos (y a menudo terminan cuando el conjunto queda vacío). Algunas aplicaciones insertan y también eliminan elementos. Se han desarrollado diversas estructuras de datos para representar estos conjuntos dinámicos. Dependiendo de las operaciones requeridas y de los patrones de acceso, diferentes estructuras de datos serán eficientes. Primero describiremos la técnica de doblado de arreglos, que es una herramienta básica. Luego presentaremos los fundamentos del análisis de tiempo amortizado, técnica que con frecuencia es necesaria para poner de manifiesto la eficiencia de implementaciones avanzadas de conjuntos dinámicos. Por último, reseñaremos varias estructuras de datos que se usan mucho y han resultado ser útiles para representar conjuntos dinámicos. Las presentaremos como implementaciones de tipos de datos abstractos (TDA) apropiados.

Los árboles rojinegros son una forma de árboles binarios equilibrados, útiles para implementar árboles de búsqueda binaria de manera eficiente. Los árboles de búsqueda binaria y las tablas de dispersión (*hash*) son implementaciones muy utilizadas del TDA Diccionario.

Se dan relaciones de equivalencia dinámicas en numerosas aplicaciones, sus operaciones están íntimamente relacionadas con el TDA Unión-Hallar, que tiene una implementación muy eficiente en ciertos casos, empleando el TDA Árbol adentro.

Las colas de prioridad son los "caballitos de batalla" de muchos algoritmos, sobre todo de los *codiciosos*. Dos implementaciones eficientes del TDA Cola de prioridad son los montones binarios (que también se usan en Heapsort) y los bosques de apareamiento, también llamados montones de apareamiento perezosos.

Este capítulo es una introducción a los temas descritos anteriormente. Si desea leer más sobre el tema o quiere leer tratamientos más extensos, consulte las Notas y referencias al final del capítulo.

6.2 Doblado de arreglos

Una situación típica que se presenta en relación con los conjuntos dinámicos es no saber de qué tamaño vamos a necesitar un arreglo cuando se inician los cálculos. Por lo regular no es muy satisfactorio reservar espacio para el arreglo "más grande que podría requerirse", aunque es una solución común. Una solución sencilla, más flexible, consiste en reservar inicialmente espacio para un arreglo pequeño con la intención de doblar su tamaño en el momento en que sea evidente que es demasiado pequeño. Para que esto funcione es preciso saber qué tan lleno está el arreglo actual y para cuántos elementos se ha reservado espacio hasta el momento. Java se mantiene al tanto del segundo dato automáticamente con el campo `length`, pero el primer dato es responsabilidad del programador y depende de la aplicación para la que se usa el arreglo.

Supóngase que tenemos una clase organizadora `arregloConjuntos` con dos campos, `tamConjunto` y `elementos`, este último es un arreglo del tipo de elementos, que suponemos es simplemente **`Object`**. En un principio, podríamos construir un objeto de esta clase como sigue:

```
arregloConjuntos miConjunto = new arregloConjuntos();
miConjunto.tamConjunto = 0;
miConjunto.elementos = new Object[100];
```

Ahora, cada vez que se añada un elemento a `miConjunto`, el programa incrementará también `tamConjunto`. Sin embargo, antes de insertar un elemento nuevo, el programa deberá cerciorarse de que haya espacio y, si no lo hay, doblar el tamaño del arreglo. Esto se hace reservando espacio para un nuevo arreglo que es dos veces más grande que el arreglo actual y transfiriendo luego todos los elementos al nuevo arreglo. El código de la aplicación podría ser el siguiente:

```
if (miConjunto.tamConjunto == miConjunto.elementos.length)
   doblarArreglo(miConjunto);
Continuar con la inserción del elemento nuevo.
```

La subrutina `doblarArreglo` tiene la forma siguiente:

```
doblarArreglo(conjunto)
   nuevaLongitud = 2 * conjunto.elementos.length;
   nuevosElementos = new Object[nuevaLongitud];
   Transferir todos los elementos del arreglo conjunto.elementos
   al arreglo nuevosElementos.
   conjunto.elementos = nuevosElementos;
```

La parte costosa es la transferencia de elementos. Sin embargo, ahora demostraremos que el procesamiento fijo total para insertar n elementos en un conjunto almacenado de esta manera está en $\Theta(n)$.

Supóngase que la inserción del $(n + 1)$-ésimo arreglo dispara una operación de doblado de arreglo. Sea t el costo de transferir un elemento del arreglo viejo al nuevo (suponemos que t es una constante). Entonces, se efectuarán n transferencias como parte de esta operación de doblado de arreglo. Sin embargo, se efectuaron $n/2$ transferencias en la operación de doblado de arreglo anterior y $n/4$ en la anterior a ésa, etc. El costo total de todas las transferencias desde que se creó el conjunto no puede exceder $2t\ n$.

Éste es un ejemplo sencillo en el que es posible *amortizar* o "pagar a plazos", el costo de operaciones ocasionales costosas, de modo que el procesamiento fijo por operación esté acotado por una constante. El análisis de tiempo amortizado se explica en la sección siguiente.

6.3 Análisis de tiempo amortizado

Como vimos en la sección anterior, pueden surgir situaciones en las que el trabajo efectuado en operaciones individuales del mismo tipo varía ampliamente, pero el tiempo total de una sucesión larga de operaciones es mucho menor que el tiempo de peor caso para una operación multiplicado por la longitud de la sucesión. Estas situaciones se presentan con relativa frecuencia en relación con los conjuntos dinámicos y sus operaciones correspondientes. Ha evolucionado una técnica llamada *análisis de tiempo amortizado* para hacer un análisis más exacto en tales situaciones. El calificativo *amortizado* proviene (en una interpretación un tanto holgada) de la práctica contable en los negocios de repartir un costo grande, en el que en realidad se incurrió en un solo periodo de tiempo, entre varios periodos de tiempo relacionados con el motivo por el que se incurrió en el costo. En análisis de algoritmos, el costo grande de una operación se reparte entre muchas operaciones, donde las otras son menos costosas. En esta sección presentamos una breve introducción al análisis de tiempo amortizado. La técnica es sencilla en lo conceptual, aunque se requiere creatividad para idear esquemas eficaces en el caso de problemas difíciles.

Supóngase que tenemos un TDA y queremos analizar sus operaciones empleando análisis de tiempo amortizado. Usamos el término *operación individual* para referirnos a una sola ejecución

de una operación. El análisis de tiempo amortizado se basa en la ecuación siguiente, que se aplica a cada operación individual del TDA en cuestión que se ejecuta en el curso de algún cálculo.

$$\textit{costo amortizado} = \textit{costo real} + \textit{costo contable.} \tag{6.1}$$

La parte creativa consiste en diseñar un sistema de *costos contables* para operaciones individuales que logre estas dos metas:

1. En *cualquier* sucesión permitida de operaciones, comenzando por la creación del objeto de TDA que se está analizando, la suma de los *costos contables* no es negativa.
2. Si bien el *costo real* podría variar ampliamente de una operación individual a la siguiente, es factible analizar el *costo amortizado* de cada operación (es decir, es relativamente regular).

Si se logran estas dos metas, el costo total *amortizado* de una sucesión de operaciones (que siempre comienza con la creación del objeto de TDA) es una cota superior del costo *real* total *y* el costo amortizado total es susceptible de análisis.

Intuitivamente, la suma de los costos contables es como una cuenta de ahorros. En épocas de bonanza, hacemos depósitos previendo un día lluvioso. Cuando llega ese día, en la forma de una operación individual inusitadamente costosa, hacemos un retiro. Sin embargo, para mantener nuestra solvencia, no podemos permitir que el saldo de nuestra cuenta se vuelva negativo.

La idea principal para diseñar un sistema de costos contables es que las operaciones individuales "normales" deben tener un costo contable positivo, mientras que a las operaciones individuales inusitadamente costosas se les asigna un costo contable negativo. El costo contable negativo deberá compensar el gasto inusitado, es decir, el costo real elevado, de modo que el costo amortizado sea aproximadamente el mismo para las operaciones individuales "normales" y para las "inusitadamente costosas". El costo amortizado podría depender del número de elementos que hay en la estructura de datos, pero deberá ser relativamente independiente de los detalles de dicha estructura. Deducir qué tan altos deben ser los cargos positivos suele requerir creatividad y podría implicar cierto grado de tanteo para llegar a una cifra que sea razonablemente pequeña, pero lo bastante grande como para evitar que el "saldo de la cuenta" se vuelva negativo.

Ejemplo 6.1 Esquema contable para Pila con doblado de arreglo

Consideremos el TDA Pila que tiene dos operaciones, `push` y `pop`, y se implementa con un arreglo. (En este ejemplo haremos caso omiso de los costos de las operaciones de acceso, ya que no modifican la pila y están en $O(1)$.) Tras bambalinas se usa doblado de arreglos, que describimos en la sección 6.2, para agrandar el arreglo en caso necesario. Digamos que el costo real de `push` o `pop` es 1 cuando no hay redimensionamiento del arreglo y el costo real de `push` es $1 + nt$, para alguna constante t, si la operación implica doblar el tamaño del arreglo de n a $2n$ y copiar n elementos en el nuevo arreglo. (El ejercicio 6.2 considera esquemas en los que tanto `push` como `pop` podrían dar pie a un redimensionamiento del arreglo.)

El tiempo real de peor caso para `push` está en $\Theta(n)$. Considerar el tiempo real de peor caso podría dar la idea de que esta implementación es muy ineficiente, puesto que son posibles implementaciones $\Theta(1)$ para estas operaciones. Sin embargo, la técnica de análisis amortizado proporciona una imagen más exacta. Podemos establecer el siguiente esquema contable:

1. El *costo contable* de un `push` que no requiere doblado del arreglo es $2t$.
2. El *costo contable* de un `push` que requiere doblar el arreglo de n a $2n$ es $-nt + 2t$.
3. El *costo contable* de un `pop` es 0.

El coeficiente de 2 en los costos contables se escogió porque es lo bastante grande como para que, desde el momento en que se crea la pila, la sumatoria de los costos contables nunca pueda ser negativa. Para comprobar esto informalmente, supóngase que habrá doblado cuando la pila alcance los tamaños N, $2N$, $4N$, $8N$, etc. Consideremos el peor caso, en el que sólo se apilan elementos. El "saldo de la cuenta" —la sumatoria neta de los costos contables— crecerá hasta $2Nt$, luego el primer cargo negativo lo reducirá a $Nt + 2t$, luego volverá a aumentar a $3Nt$ antes del segundo doblado del arreglo, después del cual bajará otra vez a $Nt + 2t$. De ahí crecerá a $5Nt$, bajará a $Nt + 2t$, crecerá a $9Nt$, se le recortará a $Nt + 2t$, y así sucesivamente. Por tanto, tenemos un esquema contable válido para el TDA Pila. (Experimentando un poco podremos convencernos de que cualquier coeficiente menor que 2 llevará tarde o temprano a la quiebra en el peor caso.)

Con este esquema contable, el *costo amortizado* de cada operación `push` individual es $1 + 2t$, sea que cause un doblado del arreglo o no, y el *costo amortizado* de cada operación `pop` es 1. Así, podemos decir que tanto `push` como `pop` se ejecutan en un *tiempo amortizado* de peor caso que está en $\Theta(1)$. ■

Las estructuras de datos más complicadas a menudo requieren esquemas contables más complicados, que a su vez requieren más creatividad. En secciones posteriores de este capítulo (secciones 6.6.6 y 6.7.2) veremos tipos de datos abstractos e implementaciones que requieren un análisis de tiempo amortizado si se quiere demostrar su eficiencia.

6.4 Árboles rojinegros

Los árboles rojinegros son árboles binarios que satisfacen ciertos requisitos estructurales. Dichos requisitos implican que la altura de un árbol rojinegro con n nodos no puede exceder $2 \lg(n + 1)$. Es decir, su altura no es más de dos veces mayor o menor que la altura del árbol binario más equilibrado que tiene n nodos. El uso más popular de los árboles rojinegros es en árboles de búsqueda binaria, pero no es la única aplicación. En esta sección mostraremos la forma de usar árboles rojinegros para mantener árboles de búsqueda binaria equilibrados (con el grado de equilibrio que acabamos de mencionar) con gran eficiencia. En las Notas y referencias al final del capítulo se mencionan algunos esquemas para mantener árboles binarios equilibrados. Hemos optado por concentrarnos en los árboles rojinegros porque el procedimiento de eliminación es más sencillo que en la mayor parte de las alternativas.

Después de introducir algo de notación, repasaremos los árboles de búsqueda binaria. Luego presentaremos las propiedades estructurales que deben tener los árboles rojinegros y mostraremos cómo mantenerlas de manera eficiente durante las operaciones de inserción y eliminación.

Los árboles rojinegros son objetos de una clase `ArbolRN`, cuya implementación con toda seguridad tendrá muchas similitudes con la del TDA `ArbolBin` de la sección 2.3.3; sin embargo, las especificaciones y la interfaz son muy distintas. Ello se debe a que un árbol rojinegro tiene un propósito más específico que un árbol binario general del TDA `ArbolBin` y tiene operaciones que modifican su estructura, mientras que no se han definido tales operaciones para el TDA `ArbolBin`. Un árbol vacío se representa con `nil` igual que en el TDA `ArbolBin`. Las operaciones con árboles rojinegros son `insertarEnArn`, `borrarDeArn` y `buscarEnArn`, que respectivamente

insertan, eliminan o buscan una clave dada en el árbol. No se proporciona acceso directo a los subárboles de un árbol rojinegro, como se hace en el TDA `ArbolBin`, aunque es posible añadir tales funciones de acceso en el entendido de que los subárboles son árboles de búsqueda binaria, pero no necesariamente árboles rojinegros.

La clase `ArbolRN` es apropiada para usarse en una implementación del TDA `Diccionario`, u otros TDA que necesiten árboles binarios equilibrados. Los nodos de un árbol rojinegro son objetos de alguna clase `Elemento`; los detalles no son importantes para los algoritmos de árbol rojinegro. Éste sería el tipo de los elementos que se almacenan en el diccionario. Aquí seguiremos muchas de las convenciones en materia de elementos, claves y comparaciones de claves que se introdujeron para los ordenamientos (capítulo 4). Suponemos que uno de los campos de la clase `Elemento` se llama `clave` y que es de la clase `Clave`. Para facilitar la notación, supondremos que las claves se pueden comparar con los operadores acostumbrados, como "$<$".

Árboles debidamente trazados

La idea de un árbol *debidamente trazado* ayuda a visualizar muchos de los conceptos relacionados con árboles de búsqueda binaria y árboles rojinegros. En este libro usaremos árboles debidamente trazados en todas las ilustraciones.

Definición 6.1

Un árbol está *debidamente trazado* en un plano bidimensional si:

1. Cada nodo es un punto y cada arista es un segmento de línea o una curva que conecta un padre a un hijo. (En un dibujo en el que los nodos son círculos o figuras similares, se considera que su "punto" está en el centro y que las aristas llegan a esos puntos.)
2. Los hijos izquierdo y derecho de cualquier nodo están a la izquierda y a la derecha, respectivamente, de ese nodo, en términos de ubicación horizontal.
3. Para cualquier arista uv, donde u es el nodo padre, ningún punto de la arista uv tiene la misma ubicación horizontal (es decir, no está directamente abajo ni directamente arriba) de cualquier antepasado propio de u. ■

En un árbol debidamente trazado, todos los nodos del subárbol izquierdo de un árbol dado están a la izquierda de la raíz y todos los nodos del subárbol derecho están a la derecha de la raíz, considerando únicamente sus ubicaciones horizontales. Si un árbol binario está debidamente trazado, una línea vertical que se desplace de izquierda a derecha se topará con los nodos en su orden de recorrido en orden interno.

Árboles vacíos como nodos externos

En los árboles de búsqueda binaria y especialemente en los árboles rojinegros, es recomendable tratar los árboles vacíos como un tipo especial de nodo, llamado *nodo externo*. Procedimos a introducir los nodos externos en relación con los árboles-2 (sección 3.4.2) y los usamos para analizar árboles de decisión (sección 4.7). En este esquema, un nodo externo no puede tener hijos y un nodo interno debe tener dos hijos. Sólo los nodos internos contienen datos, incluida una clave. En términos del TDA `ArbolBin`, podemos ver un subárbol vacío (`nil`, devuelto por las funciones `subarbolIzq` o `subarbolDer`) como una arista a un nodo externo. Todos los demás subárboles tienen como raíz un nodo interno.

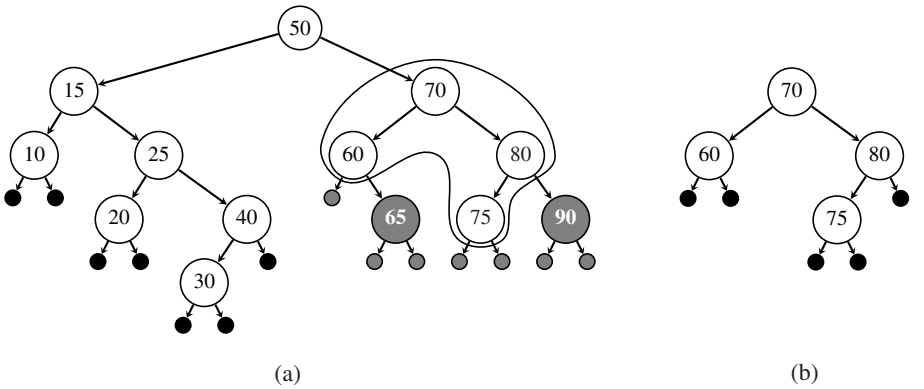


Figura 6.1 (a) Grupo de nodos con cuatro nodos, encerrados por una línea, y sus cinco subárboles principales, que aparecen en gris: los nodos pequeños denotan nodos externos. (b) El nuevo árbol *T*: los subárboles principales son sustituidos por nodos externos.

Definición 6.2 Grupos de nodos y sus subárboles principales

Un *grupo de nodos* es cualquier grupo conectado de nodos internos de un árbol binario. Un subárbol *S* es un *subárbol principal* de un grupo de nodos si el padre de la raíz de *S* está en el grupo, pero ningún nodo de *S* está en el grupo. Un subárbol principal de un grupo de nodos puede ser un nodo externo (árbol vacío). ■

La figura 6.1(a) muestra un grupo de nodos y sus subárboles principales. Podemos ver un grupo de nodos como los nodos internos de un árbol nuevo *T*, como se sugiere en la figura 6.1(b). Se extrae el grupo de nodos y se anexan nodos externos donde estaban los subárboles principales. El número de subárboles principales de un grupo de nodos siempre es uno más que el número de nodos del grupo. (¿Qué propiedad de los árboles-2 está relacionada con este hecho?)

6.4.1 Árboles de búsqueda binaria

En un árbol de búsqueda binaria, las claves de los nodos satisfacen las restricciones siguientes.

Definición 6.3 Propiedad de árbol de búsqueda binaria

Un árbol binario en el que los nodos tienen *claves* de un conjunto ordenado tiene la *propiedad de árbol de búsqueda binaria* si la clave que está en cada uno de sus nodos es mayor que todas las claves que están en su subárbol izquierdo y es menor que todas las claves que están en su subárbol derecho. En este caso decimos que el árbol binario es un *árbol de búsqueda binaria* (también se usan sus iniciales en inglés, BST). ■

Un recorrido en orden interno de un árbol de búsqueda binaria produce una lista ordenada de las claves. Si un árbol binario debidamente trazado es o no un árbol de búsqueda binaria se puede determinar fácilmente por inspección, barriendo una línea vertical de izquierda a derecha, como se mencionó en la definición 6.1. En la figura 6.2 se dan ejemplos. Como muestra esa figura, los árboles de búsqueda binaria pueden variar considerablemente en cuanto a su grado de equilibrio.

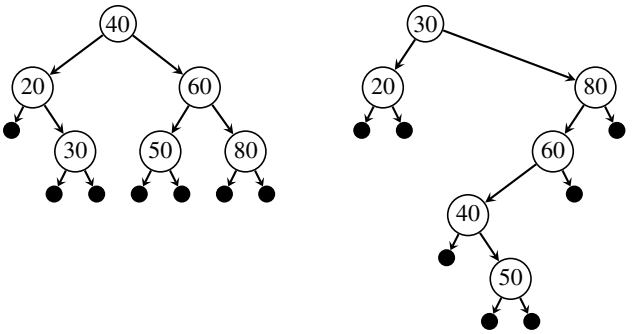


Figura 6.2 Dos árboles de búsqueda binaria con el mismo conjunto de claves pero diferentes grados de equilibrio: los puntos negros denotan árboles vacíos, también llamados nodos externos en esta sección.

Para buscar una clave dada, partimos de la raíz y seguimos la rama izquierda o la derecha dependiendo de si la clave buscada es menor o mayor que la clave que está en el nodo actual. Este procedimiento establece el patrón para todas las operaciones de BST. Las operaciones de inserción y eliminación para árboles rojinegros desarrolladas en las secciones 6.4.5 y 6.4.6 tienen incorporada la misma lógica de búsqueda.

Algoritmo 6.1 Recuperación de árbol de búsqueda binaria

Entradas: `bst`, el árbol de búsqueda binaria; y *K*, la clave buscada.

Salidas: Un objeto del árbol cuyo campo `clave` es *K*, o **`null`** si *K* no es la clave de ningún nodo del árbol.

```
Elemento buscarBst(ArbolBin bst, Clave K)
   Elemento hallado;
   if (bst == nil)
      hallado = null;
   else
      Elemento raiz = raiz(bst);
      if (K == raiz.clave)
        hallado = raiz;
      else if (K < raiz.clave)
        hallado = buscarBst(subarbolIzq(bst), K);
      else
        hallado = buscarBst(subarbolDer(bst), K);
   return hallado;
```

Usamos como medida del trabajo realizado el número de nodos internos del árbol que se examinan durante la búsqueda de la clave. (Aunque en el algoritmo, en lenguaje de alto nivel, *K* se compara con una clave del árbol dos veces, es razonable contarla como una sola comparación de tres vías, como se argumentó en la sección 1.6. De cualquier manera, el número de comparacio-

nes es proporcional al número de nodos examinados.) En el peor caso (incluidos los casos en que *K* no está en el árbol), el número de nodos examinados es la altura del árbol. (En esta sección la altura de un árbol que tiene un solo nodo interno es 1, porque los árboles vacíos se tratan como nodos externos; en la sección 2.3.3 la altura de semejante árbol se definió como 0; la diferencia no es importante en tanto se use de forma consistente la misma convención.)

Supóngase que hay *n* nodos internos en el árbol. Si la estructura del árbol es arbitraria (por lo que podría consistir en una sola cadena larga), el peor caso está en $\Theta(n)$. Si el árbol está lo más equilibrado posible, el número de nodos examinados en el peor caso es aproximadamente lg *n*. Todas las operaciones en árboles de búsqueda binaria siguen el patrón de `buscarBst` y tienen peores casos proporcionales a la altura del árbol. La meta de un sistema de árbol equilibrado es reducir el peor caso a $\Theta(\log n)$.

6.4.2 Rotaciones de árbol binario

La estructura de un árbol binario se puede modificar localmente con operaciones llamadas *rotaciones* sin perturbar la propiedad de árbol de búsqueda binaria. Aunque las operaciones de reequilibración para árboles rojinegros se pueden describir sin usar rotaciones, éstas son operaciones valiosas por derecho propio y constituyen una buena introducción a operaciones de reestructuración más complejas. De hecho, las operaciones de reestructuración más complejas se pueden armar con una sucesión de rotaciones.

En una rotación intervienen un grupo de dos nodos conectados, digamos *p* y *h*, por padre e hijo y los tres subárboles principales del grupo. En la figura 6.3 se ilustra la descripción siguiente, donde 15 hace el papel de *p* y 25 hace el papel de *h*. La arista entre *p* y *h* cambia de dirección y el subárbol principal *de enmedio* (que aparece en gris en la figura) cambia de padre, de *h* a *p*. Puesto que *h* ahora es la raíz del grupo, el antiguo padre de *p* (50 en la figura) ahora es el padre de *h*, así que debe tener una arista a *h* en vez de a *p*. Así, se modifican en total tres aristas durante una rotación.

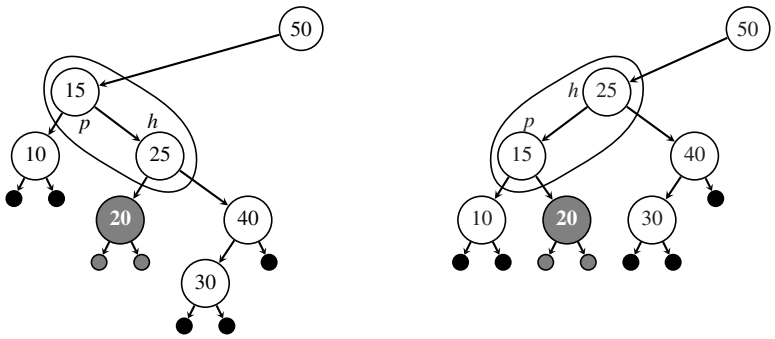


Figura 6.3 Una rotación izquierda sobre (15, 25) transforma el árbol de la izquierda en el árbol de la derecha. (No se muestra el subárbol derecho del nodo 50.) Una rotación derecha sobre (25, 15) transforma el árbol de la derecha en el de la izquierda.

En una rotación izquierda, *p* está a la izquierda de *h*, así que *p* se hunde, *h* sube y la arista al subárbol de enmedio se mueve hacia la izquierda para conectarse con *p*. El subárbol principal izquierdo se hunde junto con *p*; el subárbol principal derecho sube junto con *h*; el subárbol principal de enmedio permanece en el mismo nivel. Una rotación derecha es el inverso de una rotación izquierda; es decir, efectuar una rotación izquierda seguida de una rotación derecha sobre el mismo grupo de dos nodos deja el árbol como estaba antes. Como sugiere la figura 6.3, si las rotaciones se escogen con cuidado, el equilibrio de un árbol binario puede mejorar.

6.4.3 Definiciones de árbol rojinegro

Los árboles rojinegros son objetos de una clase `ArbolRN`. Definimos esta clase de modo que tenga cuatro campos de ejemplar, `raiz`, `subarbolIzq`, `subarbolDer` y `color`. El campo `color` especifica el color del *nodo raíz* del árbol. Aunque los nodos individuales (de clase `Elemento`) no tienen un campo `color`, todo nodo es raíz de *algún* subárbol, así que cada nodo tiene asociado un color. Para fines de implementación, definimos `color` como un campo del árbol, no del nodo, de modo que los tipos de nodos no tienen que ser específicos para árboles rojinegros. No obstante, cuando estemos hablando de árboles y nodos en términos abstractos, hablaremos del color de los nodos.

Los colores de los nodos pueden ser rojo o negro (constantes definidas en la clase). Un nodo puede ser gris temporalmente durante su eliminación, pero la estructura no será un árbol rojinegro hasta que esta condición cambie. El color de un subárbol vacío (representado por la constante `nil`, también se conoce como nodo externo) y, por definición, es negro.

Definición 6.4 Árbol rojinegro

Sea *T* un árbol binario en el que cada nodo tiene un color, rojo o negro, y todos los nodos externos son blancos. Una arista a un nodo negro es una *arista negra*. La *longitud negra* de un camino es el número de aristas negras que hay en ese camino. La *profundidad negra* de un nodo es la longitud negra del camino desde la raíz del árbol hasta ese nodo. Un camino desde un nodo específico hasta un nodo externo se denomina *camino externo* del nodo especificado. Un árbol *T* es un *árbol rojinegro* (*árbol RN* para abreviar) si y sólo si:

1. Ningún nodo rojo tiene un hijo rojo.
2. La longitud negra de todos los caminos externos que parten de un nodo dado *u* es la misma; este valor es la *altura negra* de *u*.
3. La raíz es negra.

Un árbol *T* es un *árbol casi rojinegro* (árbol CRN) si la raíz es roja, pero se cumplen las demás condiciones especificadas. ■

La figura 6.4 muestra algunos árboles rojinegros que pueden formarse con las claves de la figura 6.2. Los nodos claros son rojos. La raíz de cada árbol tiene altura negra de dos. El árbol de la extrema derecha tiene la altura máxima que puede tener un árbol rojinegro de seis nodos. Cabe señalar que su altura es menor que la del árbol de la derecha de la figura 6.2.

Podemos entender mejor la estructura de los árboles rojinegros dibujándolos de modo que los *nodos rojos estén en el mismo nivel que sus padres*. Con esta convención, la profundidad geométrica corresponde a la profundidad negra y todos los nodos externos (árboles vacíos) aparecen a

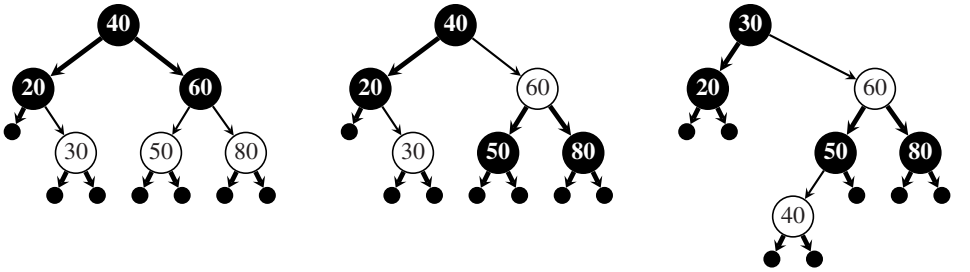


Figura 6.4 Varios árboles rojinegros con el mismo conjunto de claves: las aristas más gruesas son *aristas* negras.

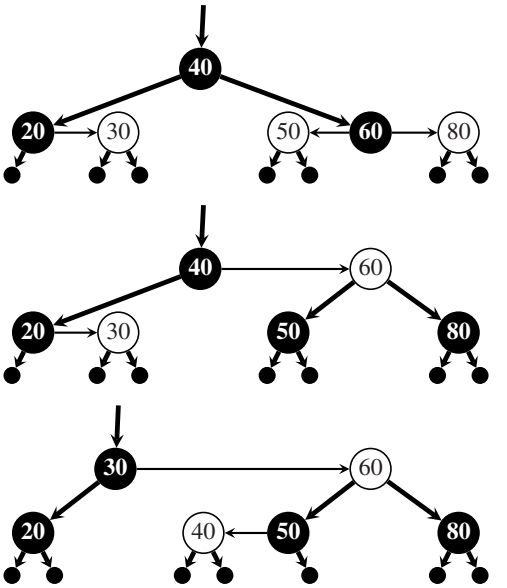


Figura 6.5 Árboles rojinegros dibujados con la convención de profundidad negra. Hemos dibujado una flecha que apunta a la raíz por claridad.

la misma profundidad. Los árboles de la figura 6.4 se han vuelto a dibujar en la figura 6.5 siguiendo esta convención, que se denomina *convención de profundidad negra*.

Examinemos ahora algunos árboles CRN. En la figura 6.5, el subárbol cuya raíz es 60 en la figura de hasta abajo es un ejemplo de árbol CRN. Si este subárbol fuera todo el árbol, bastaría con cambiar el color de la raíz a negro para tener un árbol RN. De hecho, si vemos los otros subárboles de esa figura que tienen raíz roja, constataremos que todos son árboles CRB. La definición inductiva siguiente equivale a la definición 6.4 en cuanto a que ambas definen las mismas es-

tructuras, pero la nueva definición da más detalles. Cabe señalar que un árbol RN_h es un árbol rojinegro con altura negra h.

Definición 6.5 Árboles RN_h y árboles CRN_h

Los árboles binarios cuyos nodos están coloreados rojo o negro y en los que los nodos externos son negros, son árboles RN_h y árboles CRN_h, a saber:

1. Un nodo externo es un árbol RN_0.
2. Para $h \geq 1$, un árbol binario es un árbol CRN_h si su raíz es roja y sus subárboles izquierdo y derecho son ambos árboles RN_{h-1}.
3. Para $h \geq 1$, un árbol binario es un árbol RN_h si su raíz es negra y sus subárboles izquierdo y derecho son un árbol RN_{h-1} o bien un árbol CRN_h. ■

Hacer el ejercicio 6.4 (dibujar algunos árboles RN_h y CRN_h) le ayudará a entender claramente esta definición.

Lema 6.1 La altura negra de cualquier árbol RN_h o CRN_h está bien definida y es h.

Demostración Ejercicio 6.5. □

6.4.4 Tamaño y profundidad de árboles rojinegros

Tan sólo de las definiciones, sin estudiar algoritmos, podemos deducir varios aspectos útiles de los árboles rojinegros. Es fácil demostrar por inducción estos hechos, empleando la definición 6.5, por tanto se dejan como ejercicios.

Lema 6.2 Sea T un árbol RN_h. Es decir, sea T un árbol rojinegro con altura negra h. Entonces:

1. T tiene por lo menos $2^h - 1$ nodos negros internos.
2. T tiene cuando más $4^h - 1$ nodos internos.
3. La profundidad de cualquier nodo negro es cuando más el doble de su profundidad negra.

Sea A un árbol CRN_h. Es decir, sea A un árbol casi rojinegro con altura negra h. Entonces:

1. A tiene por lo menos $2^h - 2$ nodos negros internos.
2. A tiene cuando más $\frac{1}{2}(4^h) - 1$ nodos internos.
3. La profundidad de cualquier nodo negro es cuando más el doble de su profundidad negra. □

Este lema da pie a cotas para la profundidad de cualquier nodo en términos de n, el número de nodos internos. El teorema siguiente muestra que el camino más largo en un árbol rojinegro es cuando más dos veces más largo que el camino más largo en el árbol binario más equilibrado que tiene el mismo número de nodos.

Teorema 6.3 Sea T un árbol rojinegro con n nodos internos. Entonces, ningún nodo tiene una profundidad mayor que $2 \lg(n + 1)$. En otras palabras, la altura de T en el sentido acostumbrado es cuando más $2 \lg(n + 1)$.

Demostración Sea h la altura negra de T. El número de nodos internos, n, es por lo menos el número de nodos internos negros, que es por lo menos $2^h - 1$, por el lema 6.2. Por tanto, $h \leq \lg(n + 1)$. El nodo con mayor profundidad es algún nodo externo y la profundidad negra de todos los nodos externos es h. Por el lema 6.2, la profundidad de cualquier nodo externo es entonces cuando más $2h$. □

6.4.5 Inserción en un árbol rojinegro

La definición de árbol rojinegro especifica una restricción sobre los colores y una sobre la altura negra. La idea de inserción en un árbol rojinegro consiste en insertar un nodo rojo, garantizando así que no se viole la restricción de altura negra. Sin embargo, el nuevo nodo rojo podría violar el requisito de que ningún nodo rojo tenga un hijo rojo. Podemos reparar esta violación sin infringir la restricción de altura negra si modificamos alguna combinación de colores y estructura.

La primera fase del procedimiento para insertar la clave K es básicamente la misma que se ejecuta al buscar la clave K en un BST y llegar a un nodo externo (árbol vacío) porque no se encontró la clave (véase el algoritmo 6.1). El siguiente paso consiste en sustituir ese árbol vacío por un árbol que contiene un nodo: K. La fase final, que se ejecuta durante el retorno de invocaciones recursivas, consiste en reparar cualquier violación del color. En ningún momento hay violaciones de la restricción de altura negra.

Ejemplo 6.2 Primera fase de la inserción rojinegra

Antes de examinar el algoritmo completo, consideremos lo que sucede en la fase 1 de la inserción si insertamos una clave nueva 70 en los árboles rojinegros de la figura 6.5. En los tres árboles, 70 se compara con la raíz y es mayor, así que la búsqueda baja al subárbol derecho. Luego 70 se compara con 60 y una vez más la búsqueda desciende hacia la derecha, donde 70 se compara con 80. Ahora la búsqueda se dirige a la izquierda y llega al nodo externo que es el subárbol izquierdo del nodo que contiene 80. Este nodo externo se sustituye por un nodo rojo nuevo que contiene la clave 70 y tiene dos nodos externos como hijos. La configuración del árbol superior en este momento se muestra en la figura 6.6. En los árboles inferior y de enmedio, la ubicación del nuevo nodo es similar, pero su padre es negro, por lo que no hay violación del color y el procedimiento termina. En el árbol superior (fig. 6.6) ha habido una violación del color, porque el nodo rojo 80 tiene un hijo rojo 70. Es preciso reparar esta violación para completar la operación de inserción. Volveremos a este ejemplo después de describir el método de reparación. ■

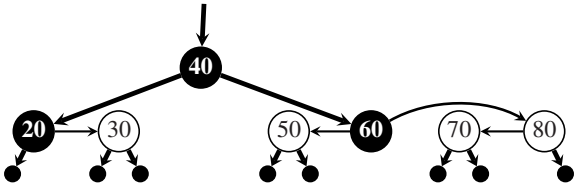


Figura 6.6 Violación de la restricción de color de los árboles rojinegros después de insertar la clave 70 en el árbol superior de la figura 6.5.

Definición 6.6 Cúmulos y cúmulos críticos

Definimos un *cúmulo* como el conjunto de nodos internos que consiste en un nodo negro y todos los nodos rojos a los que se puede llegar desde ese nodo negro siguiendo únicamente aristas no negras. (Por tanto, cada cúmulo tiene exactamente un nodo negro, la *raíz del cúmulo*.)

Si se llega a cualquier nodo de un cúmulo por un camino de longitud mayor que 1 desde la raíz del cúmulo, decimos que el cúmulo es un *cúmulo crítico*. (Puesto que todos los caminos dentro de un cúmulo constan de aristas no negras, un camino de longitud 2 implica que algún nodo rojo tiene una arista que va a otro nodo rojo.) ■

Por la definición 6.2, los *subárboles principales* de un cúmulo son aquellos subárboles cuyas raíces no están en el cúmulo, pero cuyos padres sí están en el cúmulo. Por la definición de cúmulo, las raíces de los subárboles principales de un cúmulo son negras. Un subárbol principal puede ser un nodo externo (árbol vacío).

Ejemplo 6.3 Cúmulos en árboles rojinegros

En la figura 6.6 el nodo 40 es un cúmulo, los nodos (20, 30) son un cúmulo y los nodos (60, 50, 80, 70) son un cúmulo. Este último es un cúmulo crítico porque se llega a 70 por un camino de longitud 2 desde 60, la raíz de ese cúmulo. Los subárboles principales del cúmulo 40 tienen raíces en 20 y 60. Los subárboles principales del cúmulo (60, 50, 80, 70) son cinco nodos externos y los subárboles principales del cúmulo (20, 30) son tres nodos externos. ■

Si la altura negra está bien definida para la raíz de un cúmulo, y tiene el valor h, está bien definida y es igual a h para todos los demás nodos del cúmulo, porque todos son nodos rojos. Esta altura negra está bien definida y es igual a h si y sólo si todos los subárboles principales tienen altura negra $h - 1$. Veremos que esta condición sí se cumple en todo momento durante el procedimiento de inserción.

Utilizando la terminología de cúmulos y cúmulos críticos, podemos describir en términos generales las violaciones de la definición de árbol rojinegro que podrían presentarse durante la inserción de un nodo nuevo. Si no hay cúmulos críticos en el árbol, no habrá violaciones y la operación habrá terminado. Un cúmulo crítico puede tener tres o cuatro nodos; en la figura 6.6 se muestra un ejemplo con cuatro nodos. Si no estuviera el nodo 50 (si lo sustituyéramos por un nodo externo), el cúmulo seguiría siendo crítico y tendría tres nodos.

Antes de iniciarse una operación de inserción, un árbol rojinegro no tiene cúmulos críticos (por definición). Como vimos, la fase 1 de la inserción podría crear un cúmulo crítico. Durante la reequilibración (fase 2) la estrategia consiste en reparar el cúmulo crítico haciendo que no quede ningún cúmulo crítico o bien creando un cúmulo crítico en un nivel más alto del árbol. En ningún momento hay más de un cúmulo crítico. Tarde o temprano, si la raíz del cúmulo crítico es la raíz de todo el árbol, la reparación habrá tenido éxito, así que la reequilibración tarde o temprano tiene éxito. El método de reparación depende de si el cúmulo crítico tiene tres o cuatro nodos.

Primero, consideremos un cúmulo crítico de cuatro nodos, como en la figura 6.6 para el cúmulo (60, 50, 80, 70). Realizamos una *inversión de color* con la raíz del cúmulo, a la que llamaremos r (en un principio r es negro) y sus dos hijos (que inicialmente son ambos rojos). Es decir, hacemos que la raíz r sea roja, y que los dos hijos sean negros. Esto incrementa en 1 la al-

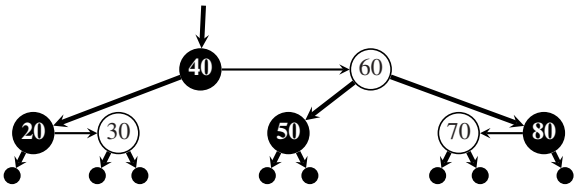


Figura 6.7 Inversión de color que repara el cúmulo crítico de cuatro nodos de la figura 6.6

tura de *r*, como se ve en la figura 6.7 donde *r* es el nodo 60. Sin embargo, la arista del padre de *r* (nodo 40 en la figura) a *r* ya no es negra, así que la longitud negra de los caminos que nacen en el padre y pasan por *r* no cambia, por tanto, la altura negra del padre sigue estando bien definida. La inversión de color repara la violación del color: el camino que era negro, rojo, rojo ahora es rojo, negro, rojo.

Si sucede que *r* es la raíz de todo el árbol, y una inversión de color hace que cambie a rojo, volverá a ser negra al término del procedimiento de inserción. (La raíz de todo el árbol también cambia a rojo cuando se inserta el primer nodo en un árbol vacío.) Las únicas ocasiones en que cambia la altura negra de todo el árbol es cuando la raíz cambia a rojo durante una inserción.

Puesto que una inversión de color hace que *r*, la raíz del antiguo cúmulo, cambie a rojo colocándola en un cúmulo distinto, existe la posibilidad de que el padre de *r* sea un nodo rojo y que el nuevo cúmulo se convierta en un cúmulo crítico. En tal caso, será preciso reparar el nuevo cúmulo crítico.

Ejemplo 6.4 Inserción en árbol rojinegro e inversiones de color

Supóngase que se insertan las claves 85 y luego 90 en el árbol de la figura 6.7. La primera inserción no causa una violación del color. La fase 1 de la segunda inserción crea la situación que se muestra en la parte superior de la figura 6.8. El cúmulo crítico consiste en (80, 70, 85, 90). La situación después de efectuar una inversión de color se muestra en la parte inferior de la figura 6.8. El nodo 80 se ha unido al cúmulo (40, 60), convirtiéndolo en un cúmulo crítico de tres nodos. Las inversiones de color no sirven en los cúmulos críticos de tres nodos (véase el ejercicio 6.7), así que necesitaremos una técnica nueva para reparar este cúmulo crítico. ■

Pasemos ahora a la técnica para reparar cúmulos críticos de tres nodos. Llamaremos a los nodos *L*, *M* y *R*, en orden de izquierda a derecha (recordemos nuestro supuesto de que el árbol se ha trazado debidamente). Este cúmulo tiene cuatro subárboles principales que llamaremos, también en orden de izquierda a derecha, *LL*, *LR*, *RL*, *RR*. Recordemos que la raíz de cada subárbol principal debe ser negra, pues de lo contrario formaría parte del cúmulo. La raíz del cúmulo crítico es *L* o bien *R*, pues de lo contrario no podría contener un camino de longitud 2. Las cuatro configuraciones posibles son los árboles (a) a (d) de la figura 6.9. Si lo vemos como un árbol de tres nodos, el cúmulo estará desequilibrado. La solución consiste simplemente en reequilibrar el cúmulo, conservando su altura negra. Es decir, *M* se convierte en la nueva raíz del cúmulo y cambia a negro; *L* se convierte en el nuevo hijo izquierdo, *R* se convierte en el nuevo hijo derecho y ambos se vuelven rojos. Ahora se reconectan los subárboles principales, conservando su orden (con lo

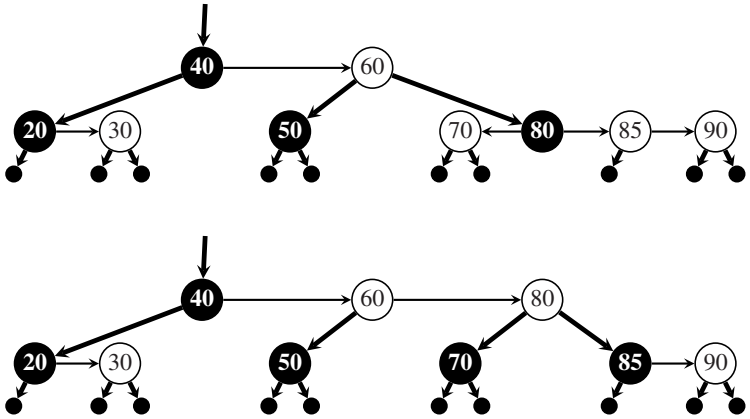


Figura 6.8 Una inversión de color repara el cúmulo crítico de cuatro nodos del árbol superior, pero produce el árbol inferior que tiene un nuevo cúmulo crítico (40, 60, 80).

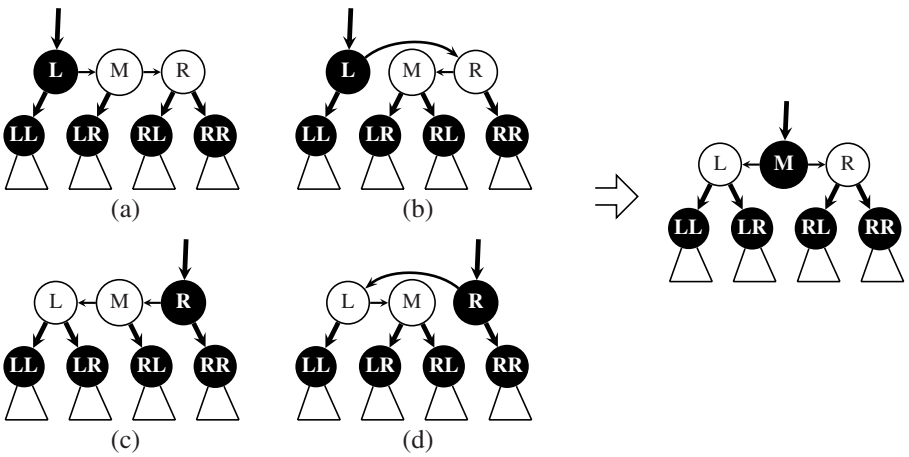


Figura 6.9 La reequilibración repara cualquier cúmulo crítico de tres nodos. Los cuatro posibles acomodos iniciales, (a) a (d), se convierten en la misma organización final, que se muestra a la derecha.

que se conserva la propiedad de árbol de búsqueda binaria). *LL* y *LR* se convierten en los hijos izquierdo y derecho de *L*, respectivamente; *RL* y *RR* se convierten en los hijos izquierdo y derecho de *R*, respectivamente. Observemos que puede haber cuatro acomodos diferentes del cúmulo antes de la reequilibración, pero sólo uno después de la reequilibración.

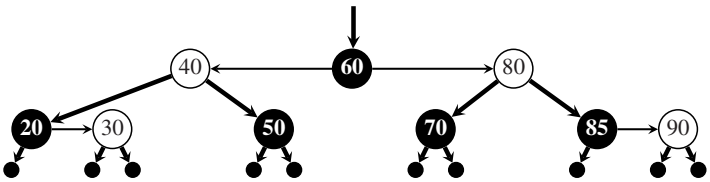


Figura 6.10 Resultado de reequilibrar el cúmulo crítico (40, 60, 80) del árbol inferior de la figura 6.8: ahora el nodo 60 es la raíz del árbol.

Ejemplo 6.5 Inserción en árbol rojinegro y reequilibración

El cúmulo crítico (40, 60, 80) del árbol inferior de la figura 6.8 se repara reequilibrando: $L = 40$, $M = 60$, $R = 80$, la raíz de *LL* es 20, la raíz de *LR* es 50, la raíz de *RL* es 70 y la raíz de *RR* es 85. Después de reequilibrar, la raíz del árbol es 60, como se muestra en la figura 6.10. ■

Ya estamos en condiciones de describir una implementación del procedimiento de inserción. En la figura 6.11 se dan las especificaciones y en la figura 6.12 se presentan los campos de ejemplar de la clase `ArbolRN`.

El procedimiento de inserción `insertarArn` emplea un procedimiento recursivo llamado `insArn`. El tipo devuelto por `insArn` es una clase organizadora, `DevuelveIns` (véase la figura 6.12) porque es deseable que las invocaciones recursivas devuelvan tanto el subárbol en el que se ha insertado el nodo nuevo como información de situación que permita detectar y reparar cualesquier violaciones.

```
ArbolRN insertarArn(ArbolRN viejoArbolRN, Elemento nuevoNodo)
```

Condición previa: `viejoArbolRN` tiene la propiedad de árbol de búsqueda binaria y satisface las propiedades de árbol rojinegro de la definición 6.4.

Condiciones posteriores: El árbol devuelto tiene `nuevoNodo` debidamente insertado. Se puede destruir `viejoArbolRN`.

```
ArbolRN borrarArn(ArbolRN viejoArbolRN, Clave K)
```

Condición previa: `viejoArbolRN` tiene la propiedad de árbol de búsqueda binaria y satisface las propiedades de árbol rojinegro de la definición 6.4.

Condiciones posteriores: Si `viejoArbolRN` no contenía ningún nodo con la clave *K*, el árbol devuelto es idéntico a `viejoArbolRN`; si `viejoArbolRN` contenía exactamente un nodo con la clave *K*, el árbol devuelto no contiene ese nodo; en los demás casos se elimina *un* nodo con la clave *K*. Se puede destruir `viejoArbolRN`.

```
Elemento buscarArn(ArbolRN T, Clave K)
```

Condición previa: T tiene la propiedad de árbol de búsqueda binaria.

Condiciones posteriores: El valor devuelto es un elemento de *T* que contiene la clave *K*, o **`nulo`** si esa clave no está en *T*.

```
ArbolRN nil
```

Constante que denota el árbol vacío.

Figura 6.11 Especificaciones de la clase `ArbolRN`

```
class ArbolRN
   Elemento raiz;
   ArbolRN subarbolIzq;
   ArbolRN subarbolDer;
   int color;


   static class DevuelveIns
      public ArbolRN nuevoArbol;
      public int situacion;
```

Figura 6.12 Campos de ejemplar privados de la clase `ArbolRN` y de la clase interna `DevuelveIns`. Además, la constante `nil` y varios métodos son públicos.

Utilizamos las constantes simbólicas siguientes, que deben definirse como valores enteros distintos en la clase `ArbolRN`. Las constantes de situación de tres letras representan los colores de los tres nodos más altos (hijo izquierdo, raíz, hijo derecho) del árbol devuelto por `insArn`.

`color`	`rojo`	el nodo raíz es rojo
	`negro`	el nodo raíz es negro
`situacion`	`ok`	operación terminada, la raíz es la misma que tenía la entrada
	`rnr`	la raíz es negra, se aplicó reparación final
	`nrn`	la raíz es roja, ambos hijos son negros
	`rrn`	la raíz y el hijo izquierdo son rojos
	`nrr`	la raíz y el hijo derecho son rojos

Algoritmo 6.2 Inserción en árbol rojinegro

Entradas: Un árbol rojinegro, `viejoArbolRN`, que también es un árbol de búsqueda binaria; `nuevoNodo`, el nodo (con clave K) que se quiere insertar. Si K es igual a una clave existente, de todos modos se insertará.

Salidas: Un árbol rojinegro con los mismos nodos que `viejoArbolRN` y además `nuevoNodo`.

Comentarios:

1. La envoltura `insertarArn` invoca al procedimiento recursivo `insArn`. Las condiciones previas y posteriores de `insArn` están contenidas en el lema 6.4.
2. Si el `nuevoArbol` devuelto por `insArn` a `insertarArn` tiene raíz roja, la envoltura la cambiará a negra.
3. Las subrutinas adicionales aparecen en las figuras 6.13 y 6.14.
4. Varias subrutinas se dejan para los ejercicios: `colorDe`, `invertirColor`, `repararDer` y `reequilDer`.

```
ArbolRN insertarArn(ArbolRN viejoArbolRN, Elemento nuevoNodo)
   DevuelveIns respuesta = insArn(viejoArbolRN, nuevoNodo);
   if (respuesta.nuevoArbol.color ≠ negro)
     respuesta.nuevoArbol.color = negro;
   return respuesta.nuevoArbol;

DevuelveIns insArn(ArbolRN viejoArbolRN, Elemento nuevoNodo)
   DevuelveIns respuesta, respIzq, respDer;
   if (viejoArbolRN == nil)
      respuesta = new DevuelveIns();
      respuesta.nuevoArbol = ArbolRN de un solo nodo con raíz roja = nuevo-
      Nodo.
      respuesta.situacion = nrn;
   else
      if (nuevoNodo.clave < viejoArbolRN.raiz.clave)
        respIzq = insArn(viejoArbolRN.subarbolIzq, nuevoNodo);
        respuesta = repararIzq(viejoArbolRN, respIzq);
      else
        respDer = insArn(viejoArbolRN.subarbolDer, nuevoNodo);
        respuesta = repararDer(viejoArbolRN, respDer);
   return respuesta;
```

Lema 6.4 Si el parámetro `viejoArbolRN` de `insArn` es un árbol RN_h o un árbol CRN_{h+1}, los campos `nuevoArbol` y `situacion` devueltos serán una de las combinaciones siguientes:

1. `situacion` = `ok` y `nuevoArbol` es un árbol RN_h o un árbol CRN_{h+1}.
2. `situacion` = `rnr` y `nuevoArbol` es un árbol RN_h.
3. `situacion` = `nrn` y `nuevoArbol` es un árbol CRN_{h+1}.
4. `situacion` = `rrn` y `nuevoArbol.color` = `rojo`, `nuevoArbol.subarbolIzq` es un árbol CRN_{h+1} y `nuevoArbol.subarbolDer` es un árbol RN_h.
5. `situacion` = `nrr` y `nuevoArbol.color` = `rojo`, `nuevoArbol.subarbolDer` es un árbol CRN_{h+1} y `nuevoArbol.subarbolIzq` es un árbol RN_h.

Demostración Ejercicio 6.12. □

Teorema 6.5 El algoritmo 6.2 inserta correctamente un nodo nuevo en un árbol rojinegro de n nodos en un tiempo $\Theta(\log n)$ en el peor caso.

Demostración La demostración es consecuencia del lema 6.4 y del teorema 6.3. Por ejemplo, si `insArn` devuelve una situación de `rrn` o `nrr`, entonces `insertarArn` cambiará el color de la raíz a negro, entonces el árbol cumplirá todas las propiedades de árbol rojinegro de la definición 6.4, pues se habrá eliminado la violación del color. Asimismo, si la situación devuelta es `nrn` u `ok`, el cambio de la raíz a negro garantizará que el árbol sea rojinegro. □

```
/** Condición previa de repararIzq:
  * viejoArbol tiene una altura negra bien definida, pero
  * podría tener dos nodos rojos consecutivos. */
/** Condición posterior: Sea respuesta el valor devuelto.
  * respuesta.nuevoArbol es el resultado de una inversión de color o ree-
    quilibración,
  * si es necesario, de viejoArbol. En los demás casos, respuesta.nue-
    voArbol = viejoArbol.
  * respuesta.situacion indica qué pasó: si se hizo una inversión de color,
  * respuesta.situacion = nrn y respuesta.nuevoArbol tiene raíz roja.
  * Si hubo reequilibración, respuesta.situacion = rnr y respuesta.nue-
    voArbol
  * es un árbol rojinegro.
  * Si no se hizo ninguna de las dos cosas, respuesta.situacion = ok y
  * respuesta.nuevoArbol es un árbol rojinegro.
  */
DevuelveIns repararIzq(ArbolRN viejoArbol, DevuelveIns respIzq)
      DevuelveIns respuesta = new DevuelveIns();
      if (respIzq.situacion == ok)
          // No hay que cambiar nada
          respuesta.nuevoArbol = viejoArbol;
          respuesta.situacion = ok;
      else
          viejoArbol.subarbolIzq = respIzq.nuevoArbol;
          if (respIzq.situacion == rnr)
            // No se requiere más reparación
            respuesta.nuevoArbol = viejoArbol;
            respuesta.situacion = ok;
          else if (respIzq.situacion == nrn)
            // Subárbol izquierdo bien; verificar color de raíz
            if (viejoArbol.color == negro)
              respuesta.situacion = ok;
            else
              respuesta.situacion = rrn;
            respuesta.nuevoArbol = viejoArbol;
          else if (colorDe(viejoArbol.subarbolDer) == rojo)
            // El cúmulo crítico es de 4.
            invertirColor(viejoArbol);
            respuesta.nuevoArbol = viejoArbol;
            respuesta.situacion = nrn;
          else
            // El cúmulo crítico es de 3.
            respuesta.nuevoArbol = reequilIzq(viejoArbol, respIzq.si-
            tuacion);
            respuesta.situacion = ok;
      return respuesta;
```

Figura 6.13 La subrutina `repararIzq` para el algoritmo 6.2

```
/** Condición previa de reequilIzq:
  * viejoArbol tiene raíz negra y altura negra bien definida, pero
  * tiene 2 nodos rojos consecutivos, como especifica situacionIzq.
  * viejoArbol.subarbolIzq es rojo en todos los casos, y uno de sus
  * hijos (nieto de viejoArbol) es rojo.
  * Si situacionIzq = rrn, es el nieto izquierdo-izquierdo.
  * Si situacionIzq = nrr, es el nieto izquierdo-derecho.
  */
/** Condición posterior: El árbol devuelto es un árbol rojinegro
  * resultado de reequilibrar viejoArbol.
  */
ArbolRN reequilIzq(ArbolRN viejoArbol, int situacionIzq)
      ArbolRN L, M, R, LR, RL;
      if (situacionIzq == rrn)  // caso (c)
         R = viejoArbol;
         M = viejoArbol.subarbolIzq;
         L = M.subarbolIzq;
         RL = M.subarbolDer;
         R.subarbolIzq = RL;
         M.subarbolDer = R;
      else
         // situacionIzq == nrr, caso (d)
         R = viejoArbol;
         L = viejoArbol.subarbolIzq;
         M = L.subarbolDer;
         LR = M.subarbolIzq;
         RL = M.subarbolDer;
         R.subarbolIzq = RL;
         L.subarbolDer = LR;
         M.subarbolDer = R;
         M.subarbolIzq = L;
      // Ahora la raíz del cúmulo es M.
      L.color = rojo;
      R.color = rojo;
      M.color = negro;
      return M;
```

Figura 6.14 Subrutina `reequilIzq` para el algoritmo 6.2: las variables `L`, `M`, `R`, `LR` y `RL` corresponden a la figura 6.9. Esta subrutina maneja los casos (c) y (d) de esa figura, donde `R` es la raíz del cúmulo crítico antes de reequilibrarlo. En el caso (c), se devolvió `rrn` del subárbol cuya raíz es `M`. En el caso (d), se devolvió `nrr` del subárbol cuya raíz es `L`. El árbol se reconfigura como se muestra a la derecha de la figura 6.9.

6.4.6 Eliminación en un árbol rojinegro

Eliminar un nodo de un árbol rojinegro es un poco más complicado que insertarlo. Por principio de cuentas, borrar un nodo de cualquier BST es más complicado que insertar un nodo en un BST. Ello se debe, intuitivamente, a que siempre es posible insertar en una hoja, pero al borrar un nodo podríamos vernos obligados a hacerlo en cualquier lugar del árbol.

Además, en el caso de un árbol rojinegro, en algunos casos será necesario restablecer el equilibrio de altura negra. Mientras que el procedimiento de inserción siempre podía mantener la altura negra correcta, sólo se tienen que reparar las violaciones del color, el procedimiento de eliminación nunca encuentra una violación del color, pero sí debe reparar errores de altura. En particular, la eliminación de un nodo negro hace que su padre quede desequilibrado. (Es decir, el padre ya no tendrá una altura negra bien definida como exige la segunda condición de la definición 6.4.) En varios casos basta cambiar el color de un nodo de rojo a negro para restablecer el equilibrio. Los casos difíciles surgen cuando no se cuenta con ese recurso.

Eliminación en un árbol de búsqueda binaria

Primero necesitamos idear un procedimiento para borrar un nodo de un BST, sin preocuparnos por que sea un árbol rojinegro. Lo que debemos recordar es que el nodo que se borrará *lógicamente*, en el sentido de que su clave desaparecerá, generalmente no es el nodo que se borra *estructuralmente*. El nodo borrado *estructuralmente* suele ser el *sucesor en el árbol* (véase la definición más adelante) del nodo borrado *lógicamente*, la información (incluida la clave) del nodo borrado estructuralmente sustituye a la del nodo borrado lógicamente. Esto no perturba el orden de las claves que exige la propiedad BST, porque la clave del sucesor en el árbol es la que sigue inmediatamente a la del nodo borrado lógicamente; dicho de otro modo, en un barrido de izquierda a derecha por un árbol debidamente trazado, el sucesor en el árbol aparece inmediatamente después del nodo que se borrará lógicamente.

Definición 6.7 Sucesor en el árbol

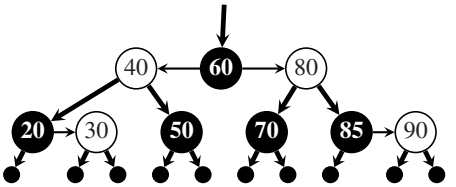
En un árbol-2, el *sucesor en el árbol* de cualquier nodo interno u es el nodo interno que está a la extrema izquierda del subárbol derecho de u o simplemente el subárbol derecho de u si es un nodo externo. ■

Si el sucesor en el árbol de u es un nodo externo, es porque u contiene la clave máxima del árbol, y bastará con borrarlo estructuralmente, subiendo su hijo izquierdo a la posición que ocupaba u. En el resto de la explicación supondremos que ésta no es la situación.

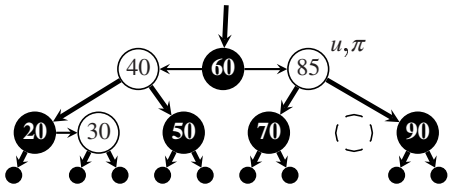
Supóngase que vamos a borrar lógicamente el nodo u. Sea σ el sucesor en el árbol de u, sea S el subárbol cuya raíz es σ y sea π el padre de σ. El subárbol izquierdo de S forzosamente está vacío, porque el sucesor en el árbol es un nodo de extrema izquierda. Por tanto, la eliminación estructural puede efectuarse conectando el subárbol derecho de S como subárbol a π, sustituyendo a S. Si $\pi = u$, entonces S era el subárbol derecho de π; si no, S era el subárbol izquierdo de π. Cabe señalar que el subárbol derecho de S podría ser un nodo externo.

Ejemplo 6.6 Eliminación en BST

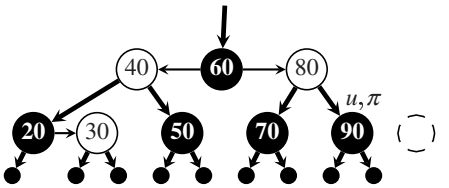
La figura 6.15 muestra varios ejemplos de borrado lógico y estructural. Aunque se incluyen los colores de los nodos para referirnos a ellos en explicaciones posteriores, no afectan el procedimiento básico de borrado en BST. En el árbol original, el sucesor en el árbol de 80 es 85, el sucesor en el árbol de 60 es 70, etcétera.



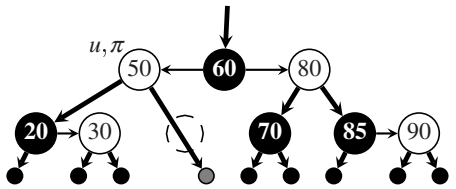
(a) Árbol original



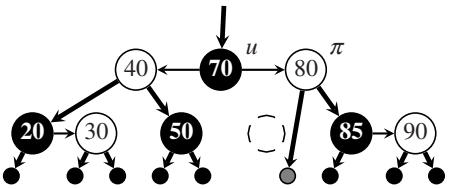
(b) Borrar 80 lógicamente



(c) Borrar 85 lógicamente



(d) Borrar 40 lógicamente



(e) Borrar 60 lógicamente

Figura 6.15 Resultado de eliminar lógicamente diversos nodos (marcados con u) en un árbol rojinegro (o cualquier árbol de búsqueda binaria con estructura de nodos similar), como se describe en los Ejemplos 6.6 y 6.7: el nodo borrado estructuralmente, σ, se indica con un círculo hecho de guiones y su expadre se marca con π. El padre del nodo gris no tiene una altura negra bien definida.

En las partes (b), (c) y (d), $\pi = u$. Es decir, el sucesor en el árbol de u es su hijo derecho. (Esto sería inusitado en un árbol más grande.) El subárbol derecho del sucesor se convertirá en el subárbol derecho de u una vez que la información del sucesor se copie en u. En términos específicos, para borrar 80 lógicamente, la información de su sucesor en el árbol, 85, se copia en el nodo que contenía a 80; luego se borra estructuralmente dicho sucesor. El subárbol cuya raíz es 90 era el subárbol derecho del antiguo sucesor en el árbol, así que ahora se convierte en el subárbol derecho de π. En el caso de borrar 85 lógicamente, la estructura se ve igual que después de borrar 80, sólo que en cada caso se borró estructuralmente un nodo distinto.

El caso más representativo se ilustra borrando 60 en la parte (e) de la figura 6.15, porque aquí u y π son nodos distintos. El sucesor en el árbol es 70. Se copia la información de 70 en u, el nodo que contenía 60; luego se borra estructuralmente el sucesor en el árbol. El subárbol derecho

del antiguo sucesor en el árbol (en este caso un nodo externo) se convierte en el subárbol izquierdo de π.

Volveremos a estos ejemplos para considerar las implicaciones de equilibrio de los árboles rojinegros. ■

Generalidades del borrado en árboles rojinegros

El procedimiento para borrar en un árbol rojinegro se puede resumir como sigue:

1. Efectuar una búsqueda en BST estándar para localizar el nodo que se borrará lógicamente, al cual llamaremos *u*.
2. Si el hijo derecho de *u* es un nodo externo, identificamos a *u* como el nodo que se borrará estructuralmente.
3. Si el hijo derecho de *u* es un nodo interno, buscamos el sucesor en el árbol de *u*, copiamos en *u* la clave y el resto de la información de dicho sucesor. (Por ahora no se cambia el color de *u*.) Identificamos el sucesor en el árbol como el nodo que se borrará estructuralmente.
4. Realizamos el borrado estructural y reparamos cualquier desequilibrio de la altura negra.

Consideremos ahora el último paso con mayor detalle.

Ejemplo 6.7 Borrado en árbol rojinegro

Demos otro vistazo a la figura 6.15. En la parte (b), aunque se borró estructuralmente un nodo negro, su hijo derecho era un nodo interno, así que forzosamente era rojo (¿por qué?), y se le podría cambiar a negro para restaurar el equilibrio de alturas negras. En la parte (c), se borró estructuralmente un nodo rojo, así que no hubo desequilibrio de altura negra.

Las partes (d) y (e) muestran el resultado cuando el sucesor en el árbol es negro y su subárbol derecho es negro (y forzosamente es un nodo externo). El subárbol restante después de la eliminación (meramente un nodo externo) no tiene suficiente altura negra, lo que se indica con el color gris. Por ejemplo, consideremos el caso en que se va a borrar 60; 70 es su sucesor en el árbol. Antes de la eliminación, el nodo 70 tenía una altura negra de 1 y era el hijo izquierdo del nodo 80. Una vez que se ha copiado 70 y se ha borrado estructuralmente el nodo que ocupaba antes (diagrama inferior derecho), el nodo externo que ocupa su lugar tiene una altura negra de 0. Ahora el árbol cuya raíz es el nodo 80 está desequilibrado con respecto a las longitudes negras de los caminos externos. La situación es similar después de borrar lógicamente el nodo 40. Éstos son ejemplos en los que se necesita una reparación que vaya más allá de un simple cambio de color. ■

Restablecimiento de la altura negra

Un nodo gris es la raíz de un subárbol que es en sí un árbol RN_{h-1}, pero que está en una posición en la que su padre requiere un árbol RN_h. (En términos más precisos, el subárbol es un árbol RN_{h-1} si interpretamos el nodo gris como negro.) Es decir, el subárbol cuya raíz es un nodo gris tiene una altura negra bien definida pero que es 1 menos que lo que se requiere para que su padre tenga una altura negra bien definida. En un principio, el nodo gris es un nodo externo, pero el color gris se podría propagar árbol arriba.

El tema de la reparación de semejante desequilibrio consiste en hallar algún nodo rojo cercano que se pueda cambiar a negro. Entonces, mediante reestructuración local, se pueden volver a

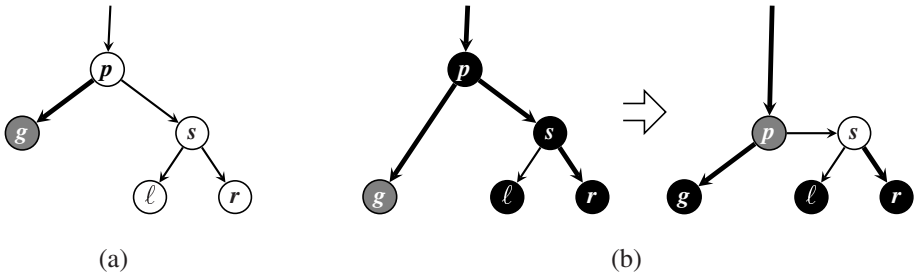


Figura 6.16 (a) Nodos en las inmediaciones de *g*, el nodo gris, cuyo árbol tiene una altura negra demasiado pequeña, durante una eliminación en un árbol rojinegro. Los colores de *p*, *s*, ℓ y *r* varían, creando diferentes casos. (b) Propagación del nodo gris cuando todos los nodos cercanos son negros. La operación es simétrica cuando *g* es un hijo derecho.

equilibrar las longitudes negras de los caminos. Si no hay ningún nodo rojo con esas características lo bastante cerca, el desequilibrio se deberá propagar a un nivel más alto del árbol y repararse recursivamente.

Vamos a introducir un poco de nomenclatura. Llamamos *g* al nodo gris, llamamos *p* a su padre y llamamos *s* a su hermano (el otro hijo de *p*). Otros dos nodos importantes son los hijos izquierdo y derecho de *s*, a los que llamaremos ℓ y *r*, respectivamente (véase la figura 6.16a).

El caso que no se puede manejar directamente es aquél en el que *p*, *s*, ℓ y *r* son todos negros. El método de propagación consiste en cambiar el color de *s* a rojo y el de *g* a negro, con lo que se reequilibra *p*, como se aprecia en la figura 6.16(b). Sin embargo, ello reduce la altura negra de *p* en 1, en comparación con el valor que tenía antes de iniciarse el borrado, así que *p* es ahora el nodo gris. Éste es el único caso en el que el nodo gris se propaga árbol arriba, e implica únicamente cambios de color, no cambios estructurales del árbol.

Cabe señalar que, si *p* es la raíz de todo el árbol y se vuelve gris, no tendrá padre al cual desequilibrar, por tanto se habrán restaurado las propiedades rojinegras (una vez que se *p* se coloree de negro). En consecuencia, no hay problema si el nodo gris se propaga árbol arriba hasta la raíz del árbol. Ahora nos concentraremos en los casos en que el nodo gris *no* se propaga.

Si cualquiera de *p*, *s*, ℓ o *r* es rojo, el desequilibrio de altura negra se podrá reparar sin propagación. Los casos se complican y son numerosos, pero tienen un tema común. Recordemos que *g* es la raíz de un árbol RN_{h-1}. Formamos un grupo de nodos con raíz en *p*, tal que todos los subárboles principales del grupo (definición 6.2) sean árboles RN_{h-1}. Llamaremos a este grupo el *grupo de reequilibración del borrado*. Ahora reestructuramos el grupo de reequilibración del borrado (aislado del resto del árbol, poniendo nodos externos en vez de los subárboles principales) como sigue:

1. Si *p* era rojo, el grupo deberá formar un árbol RN_1 o CRN_2;
2. Si *p* era negro, el grupo deberá formar un árbol RN_2.

Ahora tomamos el grupo reestructurado y lo conectamos como subárbol al padre de *p*, sustituyendo la antigua arista a *p* por una arista a la (posiblemente) nueva raíz del grupo; además, reconectamos todos los subárboles principales, en el orden correcto.

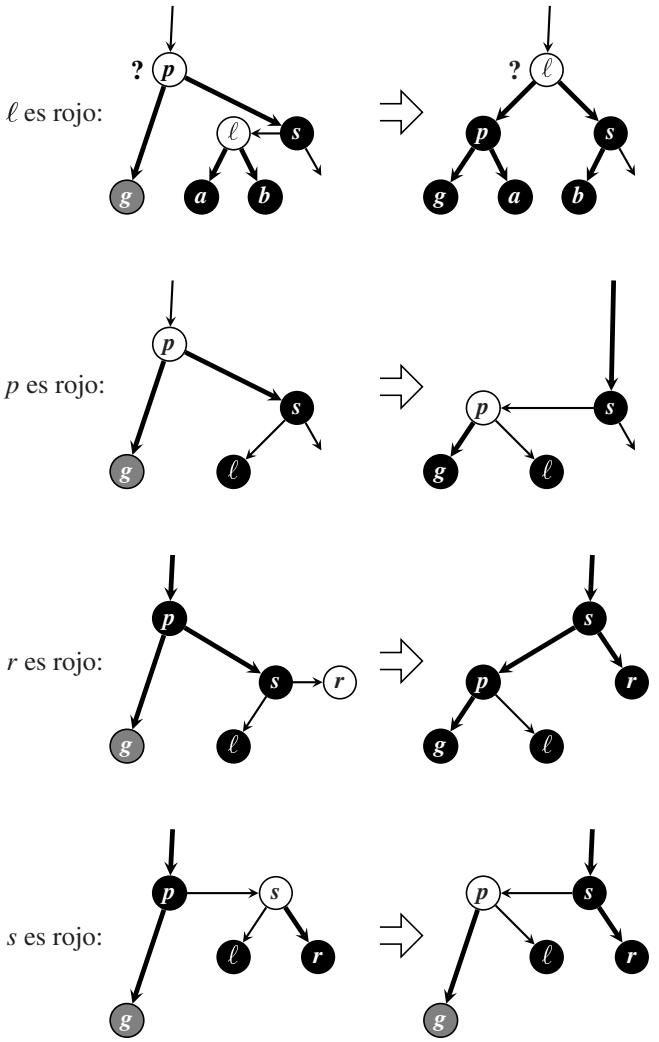


Figura 6.17 Reparaciones de errores de altura negra durante la eliminación: se consideran los casos en orden de arriba a abajo. El último caso no elimina el nodo gris, pero se convierte en uno de los dos primeros casos. Los casos son simétricos cuando g es el hijo derecho de p y entonces el orden es r, p, ℓ, s.

El número de casos se puede reducir siguiendo cierto orden al buscar nodos rojos: si g es un hijo izquierdo, el orden es ℓ, p, r y por último s. La figura 6.17 muestra las transformaciones apropiadas; cabe señalar que sólo se muestra la parte pertinente de cada grupo de reequilibración del borrado. Por ejemplo, en el primer caso, donde ℓ es rojo, el hijo derecho de s está *en* el grupo de reequilibración del borrado si es rojo, pero es la raíz de un subárbol principal del grupo si es negro. No obstante, la transformación apropiada es la misma en ambos casos.

Un signo de interrogación junto a un nodo implica que podría ser rojo o negro, pero si dos nodos tienen signo de interrogación, uno antes y uno después de la transformación, deben ser del mismo color.

En el último caso, en el que sólo *s* es rojo, no se elimina el nodo gris, pero el caso se transforma en uno de los dos primeros casos, dependiendo del color del hijo izquierdo de ℓ. (El nodo ℓ se llamará *s*, el hermano de *g*, para la reestructuración final.)

Cuando *g* es un hijo derecho, el orden es simétrico: se verifican *r*, *p*, ℓ y por último *s*. (Para que sea más fácil recordar el orden, *s* siempre es el último, y los tres primeros están en orden alfabético cuando *g* es un hijo izquierdo y en orden alfabético inverso cuando *g* es un hijo derecho.)

Ejemplo 6.8 Reparación de altura negra

Consideremos el árbol de la parte superior de la figura 6.18(a), que es el resultado de la eliminación lógica de 60 en la figura 6.15(e). El borrado creó un nodo gris. Ahora el nodo 80 hace las veces de *p*, 85 es *s*, 90 es *r* y el hijo externo (izquierdo) de 85 es ℓ. El caso aplicable de la figura 6.17 es aquel en el que *p* es rojo (y ℓ es negro), es decir, el segundo caso. Por tanto, 80 desciende al nivel de 85, que es su nuevo padre; 80 toma el antiguo hijo izquierdo de 85 como su nuevo hijo derecho. Cuando el grupo se reconecte al árbol, 70 tendrá 85 como hijo derecho en lugar de 80. El árbol final se muestra en la parte inferior de la figura 6.18(a).

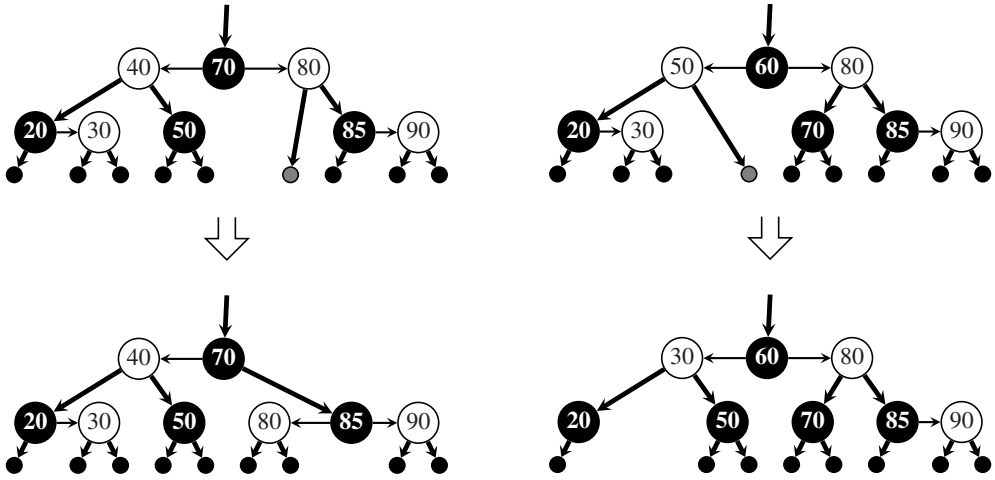
El árbol de la parte superior de la figura 6.18(b) es el resultado de la eliminación lógica de 40 en la figura 6.15(d). En este caso el nodo gris es un hijo derecho, por lo que necesitamos las "imágenes en el espejo" de la figura 6.17; obsérvese que ℓ y *r* intercambian sus papeles. El nodo 50 hace las veces de *p*, 20 es *s*, 30 es *r* y el hijo externo de 20 es ℓ. Por tanto, aplica el *primer* caso de la figura 6.17 (*r* es rojo); 30 sube al nivel en el que 50 estaba antes y adopta su color, mientras que 50 baja un nivel y se vuelve negro. Los antiguos hijos de 30 se reparten a 20 y 50. Cuando el grupo se reconecta, 30 se convierte en el nuevo hijo izquierdo de 60. El árbol final se muestra en la parte inferior de la figura 6.18(b). ■

Al igual que en las reparaciones después de una inserción, las reparaciones después de un borrado efectúan $O(1)$ cambios estructurales, pero podrían realizar $O(\log n)$ cambios de color. La implementación es tediosa debido al gran número de casos, pero no es complicada y se deja como ejercicio.

6.5 Hashing (dispersión)

El hashing o dispersión es una técnica que se usa a menudo para implementar un TDA de Diccionario, aunque también tiene otros usos. Imaginemos que fuera posible asignar un índice de arreglo único a todas las claves que pudiera haber en una aplicación. Entonces, hallar, insertar y eliminar elementos podría ser muy fácil y rápido.

Claro que, normalmente, el espacio de claves (el conjunto de todas las posibles claves) es excesivamente grande. Un ejemplo típico es el espacio de claves de cadenas de caracteres, digamos nombres. Supóngase que un nombre puede tener cuando más 20 letras y espacios. El espacio de claves tendría más de 2^{100} elementos. Es decir, si estamos usando un arreglo, necesitaría tener 2^{100} celdas para asignar un índice distinto a cada cadena, lo cual es totalmente impráctico. Aunque el espacio de claves es extremadamente grande, en una aplicación específica sólo se dará una fracción diminuta de todas las posibles claves. El conjunto real de elementos empleados podría incluir



(a) Caso 2: p(80) era rojo; ℓ (el hijo izquierdo de 85) era negro.

(b) Caso 1: r(30) era rojo; p(50) cambió de color.

Figura 6.18 Resultado de reequilibrar después de eliminar nodos, según los casos de la figura 6.17: cabe señalar que la parte (b), que es un ejemplo del caso 1, tiene su nodo gris como hijo *derecho* de p, por lo que la operación es una imagen en el espejo del caso 1 de la figura 6.17.

varios centenares, o incluso llegar a unos cuantos millones. Un arreglo con 4 millones de celdas sería suficiente para asignar un índice distinto a cada elemento y en la práctica son factibles arreglos de semejante tamaño.

El objetivo del hashing es traducir un espacio de claves extremadamente grande en un intervalo razonablemente pequeño de enteros. El valor traducido de la clave se denomina *código de dispersión* de esa clave, y se calcula mediante alguna *función de dispersión*. Podemos usar un arreglo para almacenar cada elemento según su código de dispersión.

En inglés se usa el nombre "hash" (picadillo) porque en un principio la técnica consistía en "hacer pedacitos" la clave y seleccionar ciertos bits para formar el código de dispersión de esa clave.

Lo que hace la función de dispersión es asignar enteros a claves de manera tal que sea poco probable que se asigne el mismo entero a dos claves distintas de un conjunto "típico" de n elementos. Cuando llega a suceder esto, se dice que hubo una *colisión*. Para reducir la probabilidad de colisiones, si tenemos n elementos, generalmente usamos un intervalo de enteros de hasta $2n$ para los códigos de dispersión.

El uso más común del hashing, aunque no el único, es el mantenimiento de una *tabla de dispersión*. Dicha tabla es un arreglo H con índices $0, \ldots, h - 1$; es decir, la tabla tiene h elementos. Los elementos de H se denominan *celdas de dispersión*. La función de dispersión establece una correspondencia entre cada clave y un entero del intervalo $0, \ldots, h - 1$.

Ejemplo 6.9 Hashing

Como ejemplo pequeño, supóngase que el espacio de claves consiste en enteros de cuatro dígitos, y queremos traducirlos a los enteros $0, \ldots, 7$. Escogemos la función de dispersión:

$$\texttt{codigoDisp}(x) = (5x \bmod 8).$$

Supóngase que nuestro conjunto real consiste en seis fechas históricas importantes: 1055, 1492, 1776, 1812, 1918 y 1945. Se establece la correspondencia con el intervalo 0, …, 7 así:

código de dispersión clave	0	1	2	3	4	5	6	7
	1776			1055	1492 1812	1945	1918	

Si tenemos una tabla de dispersión que consiste en un arreglo de ocho elementos, los datos pueden almacenarse según su código de dispersión y estarán repartidos por toda la tabla. Sin embargo, algunos elementos tienen el mismo código de dispersión, por lo que deben tomarse providencias para esa posibilidad. En este ejemplo, las claves 1492 y 1812 *chocaron*, lo que significa que se les hizo corresponder con el mismo código de dispersión. ■

Los dos aspectos que debemos resolver al diseñar una tabla de dispersión son: ¿cuál es la función de dispersión y cómo se manejan las colisiones? Estos dos aspectos son relativamente independientes. La "bondad" de una función como función de dispersión podría depender de la aplicación. Examinaremos el problema de las colisiones.

6.5.1 Hashing de dirección cerrada

El *hashing de dirección cerrada*, también llamado *hashing encadenado*, es la política más sencilla para manejar colisiones. Cada elemento de la tabla de dispersión, digamos $H[i]$, es una lista ligada (véase la sección 2.3.2) cuyos elementos tienen el código de dispersión i. En un principio, todos los elementos de H son listas vacías. Para insertar un elemento, primero calculamos su código de dispersión, digamos i, y luego insertamos el elemento en la lista ligada $H[i]$. Si la tabla H contiene actualmente n elementos, su *factor de carga* se define como $\alpha = n/h$. Obsérvese que α es el número promedio de elementos que hay en una lista ligada.

Para buscar una clave dada K, primero calculamos su código de dispersión, digamos i, luego examinamos la lista ligada que está en $H[i]$, comparando las claves de los elementos de la lista con K. No podemos suponer que, por el simple hecho de que el código de dispersión de un elemento es i, la clave de ese elemento es K. La función de dispersión es una función de muchos a uno.

Supóngase que es igualmente probable que tengamos que buscar cualquier elemento de la tabla, y que en ella se han almacenado n elementos. ¿Qué costo medio tiene una búsqueda exitosa? Supóngase que el costo de calcular el código de dispersión es igual al costo de efectuar un número pequeño, digamos a, de comparaciones de claves. Si el hashing coloca a un elemento en la celda i, cuya lista ligada tiene L_i elementos, entonces el número medio de comparaciones necesarias para hallar el elemento será $(L_i + 1)/2$. En tal caso, el costo medio de una búsqueda exitosa estará dado por

$$a + \frac{1}{h}\sum_{i=0}^{h-1}(L_i + 1)/2.$$

Para el esquema del ejemplo 6.9, esta cifra es $a + 7/6$. Se efectuaría un total de siete comparaciones de claves para localizar cada elemento una vez.

Si alguna fracción fija de los elementos, digamos $n/10$, se dispersa a la misma celda, una búsqueda exitosa requerirá en promedio más de $n/200$ comparaciones de claves. En el peor caso, *todos* los elementos se dispersarán a la misma celda y una búsqueda exitosa, requerirá en promedio $n/2$, o $\Theta(n)$, comparaciones de claves. Tales casos no son mejores (en cuanto a tasa de crecimiento) que la búsqueda en un arreglo no ordenado, como en el algoritmo 1.1, y ponen de manifiesto la importancia de repartir los códigos de dispersión de manera relativamente uniforme entre todo el intervalo de h enteros.

Si suponemos que los códigos de dispersión para todas las claves de nuestro conjunto tienen la misma posibilidad de ser un entero dentro del intervalo $0, \ldots, h - 1$, podremos demostrar que una búsqueda exitosa requiere en promedio $O(1 + \alpha)$ comparaciones de claves, donde $\alpha = n/h$ es el factor de carga. (En las Notas y referencias al final del capítulo se citan fuentes que tienen todos los resultados analíticos que no se deducen en el texto.) Si h es proporcional a n (lo cual puede lograrse mediante doblado del arreglo, como se describió en la sección 6.2), se requerirán en promedio $O(1)$ comparaciones de claves en una búsqueda exitosa.

En una situación práctica es poco probable que podamos justificar rigurosamente la afirmación de que los códigos de dispersión tienen una distribución uniforme. No obstante, si escogemos bien las funciones de dispersión, la experiencia apoya este supuesto en muchos casos.

Consideremos ahora el costo de una búsqueda fallida de una clave K que se dispersa al índice i. Es evidente que el peor caso es proporcional a la lista más larga de la tabla de distribución, y el promedio depende de la distribución supuesta de las solicitudes de búsqueda fallidas. Los costos de las búsquedas fallidas suelen ser peores en un factor de uno o dos que los costos de las búsquedas que sí tienen éxito.

Además de buscar una clave para atender una solicitud de recuperación, las otras operaciones que debemos considerar son la inserción y el borrado. Es evidente que la inserción no implica comparaciones de claves, depende únicamente del costo de calcular el código de dispersión. El costo del borrado es proporcional al costo de una búsqueda exitosa si el borrado tiene éxito (sólo se borra una clave en caso de haber claves repetidas), y es proporcional al costo de una búsqueda fallida si no tiene éxito.

En lugar de una lista ligada en cada celda de dispersión, ¿por qué no usar un árbol de búsqueda binaria equilibrado? Aunque ello tendría ventajas teóricas, casi nunca se hace porque, en la práctica, los factores de carga se mantienen bajos, y se confía en obtener algo parecido al comportamiento favorable de los códigos de dispersión uniformemente distribuidos. Por ello, generalmente no se considera justificable incurrir en el gasto extra de espacio y tiempo de las estructuras de datos más complejas.

6.5.2 Hashing de dirección abierta

El hashing de dirección abierta es una estrategia para almacenar todos los elementos en el arreglo de la tabla de dispersión, en vez de usar listas ligadas para dar cabida a las colisiones. Así pues, $H[i]$ contiene una clave, no una lista de claves. El direccionamiento abierto es más flexible que el cerrado porque no puede haber factores de carga mayores que 1. Por otra parte, el espacio casi siempre se aprovecha de manera más eficiente porque no se usan listas ligadas (pero véase el ejercicio 6.19). Las búsquedas se efectúan en la tabla de dispersión, sin necesidad de recorrer listas ligadas, por lo que también es más probable que la eficiencia sea mayor en términos de tiempo.

La idea fundamental del direccionamiento abierto es que, si la celda de dispersión correspondiente al código de dispersión ya está ocupada por otro elemento, se definirá una sucesión de ubicaciones alternas para el elemento actual. El proceso de calcular ubicaciones alternas se denomina *rehashing*.

La política de rehashing más sencilla es el *sondeo lineal*. Supóngase que una clave K se dispersa a la posición i, y que $H[i]$ está ocupado por alguna otra clave. Se usa la función siguiente para generar ubicaciones alternas:

$$\texttt{rehash}(j) = (j + 1) \bmod h$$

donde j es la posición sondeada más recientemente. En un principio, $j = i$, el código de dispersión para K. Cabe señalar que esta versión de `rehash` no depende de K.

Ejemplo 6.10 Sondeo lineal

Consideremos la política de sondeo lineal para almacenar las claves dadas en el ejemplo 6.9. Supóngase que las claves se insertan en el orden dado: 1055, 1492, 1776, 1812, 1918, 1945.

1. 1055 se dispersa a 3 y se almacena en $H[3]$.
2. 1492 se dispersa a 4 y se almacena en $H[4]$.
3. 1776 se dispersa a 0 y se almacena en $H[0]$.
4. 1812 se disperas a 4, pero $H[4]$ está ocupada, por lo que el sondeo lineal redispersa 4 a 5, que está vacía, así que 1812 se almacena en $H[5]$.
5. 1918 se dispersa a 6 y se almacena en $H[6]$.
6. 1945 se dispersa a 5, pero $H[5]$ está ocupada. Observemos que $H[5]$ *no* está ocupada por una clave que se haya dispersado a 5. Esto muestra que si se usa direccionamiento abierto puede haber colisiones entre claves con códigos de dispersión distintos. Sin embargo, dado que $H[5]$ está ocupada, el sondeo lineal redispersa 5 a 6 como siguiente posición en la que se intentará almacenar 1945. Esta celda también está ocupada, así que 6 se redispersa a 7, que por fin alberga a 1945.

La organización final del arreglo H es la siguiente:

índice	0	1	2	3	4	5	6	7
H	1776			1055	1492	1812	1918	1945

■

El procedimiento de recuperación imita aproximadamente el de inserción. Para buscar la clave K, se calcula su código de dispersión, digamos i. Si $H[i]$ está vacía, K no está en la tabla. Por otra parte, si $H[i]$ contiene una clave distinta de K, se hace un rehashing a $i_1 = ((i + 1) \bmod h)$. Si $H[i_1]$ está vacía, K no está en la tabla. Por otra parte, si $H[i_1]$ contiene una clave distinta de K, se hace un rehashing a $i_2 = ((i_1 + 1) \bmod h)$, y así sucesivamente.

Ejemplo 6.11 Factor de carga alto con sondeo lineal

Consideremos la búsqueda de cada una de las claves de la tabla creada en el ejemplo 6.10. Las claves 1055, 1492, 1776 y 1918 se encuentran al primer "sondeo"; es decir, están en la primera celda inspeccionada, la correspondiente a su código de dispersión. La clave 1812 requiere dos sondeos y 1945, tres. Así pues, el total de sondeos o comparaciones de claves, para el conjunto es 9, en comparación de 7 con la política de direccionamiento cerrado.

Supóngase ahora que buscamos la clave 1543, que no está en la tabla. Esta clave se dispersa a 3, así que examinamos $H[3]$, la cual no contiene 1543. El sondeo lineal redispersa 3 a 4, 4 a 5,

5 a 6 y 6 a 7, pero en cada ocasión la celda de dispersión está ocupada por una clave distinta. ¿Ya terminamos? ¡No! El rehashing es "circular". El siguiente sondeo da $((7 + 1) \bmod 8) = 0$, y $H[0]$ está ocupada por una clave distinta. Por último, 0 se redispersa a 1, donde hay una celda vacía. Esto confirma que 1543 no está en la tabla. (El lector deberá verificar por qué no es necesario examinar $H[2]$.)

Este ejemplo ilustra el punto débil del direccionamiento abierto con sondeo lineal cuando el factor de carga es cercano a 1. Se forman largas cadenas de claves con diferente código de dispersión, lo que hace necesario viajar mucho para encontrar una celda vacía. ■

Como el lector tal vez sospeche por el ejemplo anterior, incluso con el supuesto favorable de que todos los códigos de dispersión tienen la misma posibilidad de presentarse entre los elementos del conjunto, el costo medio de una búsqueda exitosa no es proporcional a α, el factor de carga, cuando se usa la política de direccionamiento abierto con sondeo lineal. De hecho, si usamos matemáticas de alto nivel, podemos demostrar que se acerca a $\sqrt{n}$ cuando el factor de carga es 1. (Véanse las Notas y referencias al final del capítulo.)

Con todos estos problemas acechando, ¿por qué habríamos siquiera de considerar el direccionamiento abierto? Una razón es que el desempeño es muy bueno cuando el factor de carga es bajo. Por ejemplo, con doblado de arreglos, es posible mantener siempre el factor de carga por debajo de 0.5. Con un factor de carga semejante es poco probable que se formen cadenas largas.

Ejemplo 6.12 Expansión de una tabla de dispersión

Consideremos otra vez las claves del ejemplo 6.9: 1055, 1492, 1776, 1812, 1918 y 1945. Supóngase que la tabla de dispersión se dobló a 16 elementos y que la nueva función de dispersión es `codigoDisp`$(x) = (5x \bmod 16)$. Ahora la correspondencia de códigos de dispersión es:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	1776				1492	1812	1918					1055		1945		

Todas las claves se guardan en las celdas correspondientes a su código de dispersión con excepción de 1812, que sigue chocando con 1492. La cadena anterior de seis celdas contiguas llenas se ha dividido en cuatro cadenas individuales. ■

Otro motivo por el que el direccionamiento sigue siendo un método eficaz es que un esquema de rehashing más avanzado alivia el problema que representan las cadenas largas de celdas ocupadas cuando el factor de carga es moderado, digamos hasta 0.7. Uno de esos esquemas es el *hashing doble*. En lugar de que `rehash` incremente en 1, como en el sondeo lineal, incrementa en d, cifra que se calcula a partir de la clave K. Es decir, calculamos $d =$ `increHash`(K) utilizando una función de dispersión distinta de `codigoDisp` y luego calculamos

$$\texttt{rehash}(j, d) = (j + d) \bmod h.$$

Así pues, si el código de dispersión de K es i y el incremento es d, la sucesión de celdas en las que se buscará es i, $(i + d)$, $(i + 2d)$, etc. El procedimiento de búsqueda en pseudocódigo sería similar al que sigue, suponiendo que la constante `celdaVacia` denota una celda de dispersión vacía.

```
Elemento hallarHash(Clave K)
   Elemento respuesta;
   int codigo = codigoDisp(K);
   int increm = increHash(K);
   int posic = codigo;
   respuesta = null;  // Se fracasa por omisión
   while (H[posic] ≠ celdaVacia)
      if (H[posic].clave == K)
           respuesta = H[posic];
           break;
      posic = rehash(posic, increm);
      if (posic == codigo)
        break;
   return respuesta;
```

El segundo **break** evita un ciclo infinito; sería innecesario si d y h se escogen de modo que la sucesión generada por `rehash` visite tarde o temprano todas las celdas del arreglo y si se sabe que el arreglo tiene por lo menos una celda vacía.

Borrado bajo régimen de direccionamiento abierto

Otra complicación se presenta si está permitido borrar elementos de una tabla de dispersión. El procedimiento de búsqueda deja de buscar cuando encuentra una celda vacía. Examinemos la organización de la tabla del ejemplo 6.10 y recordemos que 1945 se dispersa a 5 en este ejemplo, pero que debido al rehashing se almacenó en la celda 7. Supóngase que posteriormente se borra 1918, y se vuelve a asignar a H[6] el valor `celdaVacia`. Si ahora se busca 1945, la búsqueda se iniciará en 5, se redispersará a 6 y terminará en un fracaso. La clave 1945 ha quedado "aislada" de la celda correspondiente a su código de dispersión.

La forma más sencilla de evitar este problema consiste en definir otra constante, `obsoleta`. Cuando se elimina 1918, se asigna a H[6] el valor `obsoleta`. Ahora el procedimiento de búsqueda continuará, pasando por alto H[6] como si contuviera un elemento, pero *no* intentará comparar la clave de búsqueda K con esta celda. Por otra parte, la celda "obsoleta" *podría* reutilizarse para almacenar otro elemento si se presenta la ocasión. Al estimar el factor de carga, las celdas "obsoletas" cuentan como celdas llenas. Si el número de celdas "obsoletas" se vuelve excesivo con el paso del tiempo, podría ser aconsejable "hacer limpieza", reservando espacio para una nueva tabla de dispersión (vacía) y recorriendo en orden el arreglo viejo reinsertando todos los elementos válidos en la nueva tabla de dispersión.

6.5.3 Funciones de dispersión

Como hemos visto, el criterio principal de la bondad de una función de dispersión es que reparta las claves de forma relativamente uniforme. En las Notas y referencias al final del capítulo se mencionan trabajos teóricos sobre el tema. En esta sección presentaremos algunas "recetas" sencillas.

Intuitivamente, una forma de juzgar si una función dispersa bien las claves es preguntar si su salida es "predecible". Lo contrario de la predecibilidad es la aleatoriedad, así que una estrategia sencilla para escoger una función de dispersión es ajustarla al patrón de un generador de números pseudoaleatorios. Una clase de tales generadores es la de los calificados como "multiplicativos

congruenciales". En palabras con menos sílabas, esto significa "multiplicar por una constante y luego obtener el residuo después de dividir entre otra constante". La segunda constante se denomina *módulo*. En el caso de una función de dispersión, el módulo es h, el tamaño de la tabla de dispersión.

Cuando las claves son cadenas, es probable que el cálculo del código de dispersión sea el costo dominante en una operación de búsqueda o de inserción, porque por lo regular intervienen todos los caracteres de la cadena. (En la mayor parte de las comparaciones de cadenas sólo es necesario verificar uno o dos caracteres antes de hallar una diferencia.) Considerando este hecho, nuestra receta es la siguiente:

1. Escoger h como una potencia de 2, digamos 2^x, y $h \geq 8$.
2. Implementar "mod" extrayendo los x bits de orden más bajo. (El código en Java, C o C++ puede ser "(`num & (`$h - 1$`))`" porque h es una potencia de 2.)
3. Escoger el multiplicador $a = 8\lfloor h/23 \rfloor + 5$.
4. Si el tipo de la clave es entero, la función de dispersión es

$$\texttt{codigoDisp}(K) = (a\ K) \bmod h.$$

5. Si el tipo de la clave es un par de enteros (K_1, K_2), la función de dispersión puede ser

$$\texttt{codigoDisp}(K_1, K_2) = (a^2\ K_1 + a\ K_2) \bmod h.$$

6. Si el tipo de clave es una cadena de caracteres, se le trata como una sucesión de enteros, k_1, k_2, ..., y se usa como función de dispersión:

$$\texttt{codigoDisp}(K) = (a^{\ell}\ k_1 + a^{\ell-1}\ k_2 + \cdots + a\ k_l) \bmod h \qquad \text{donde } \ell \text{ es la longitud de } K.$$

   Usamos la identidad

$$(a^{\ell}\ k_1 + a^{\ell-1}\ k_2 + \cdots + a\ k_{\ell}) = ((\cdots(((k_1\ a) + k_2)\ a) + \cdots + k_{\ell})\ a)$$

   para hacer más eficiente el cálculo. Podría ser conveniente obtener mod después de cada multiplicación para evitar un desbordamiento.
7. Usamos doblado del arreglo cada vez que el factor de carga sube demasiado, digamos más allá de 0.5. Después de reservar espacio para un nuevo arreglo para la tabla de dispersión de tamaño 2^{x+1}, establecemos las constantes h y a para la nueva función de dispersión. Luego recorremos en orden el arreglo viejo y, para cada celda que contenga una clave genuina, insertamos esa clave en la nueva tabla de dispersión empleando la nueva función de dispersión.
8. Si se desea usar hashing doble, la segunda función de dispersión (llamada `increHash` en el procedimiento de búsqueda de la sección 6.5.2) puede ser más sencilla, a fin de ahorrar tiempo. Por ejemplo, si el tipo de las claves es cadena de caracteres, usamos $(2\ k_1 + 1) \bmod h$. Al calcular un incremento impar garantizamos que se accederá a toda la tabla de dispersión en la búsqueda de una celda vacía (siempre que h sea una potencia de 2).

Como dijimos, ésta es una receta para poner en marcha una tabla de dispersión con un mínimo de trabajo, lo cual es útil para implementar un TDA de diccionario.

6.6 Relaciones de equivalencia dinámicas y programas Unión-Hallar

Las relaciones de equivalencia dinámicas se presentan en diversos problemas de conjuntos o grafos. El tipo de datos abstracto Unión-Hallar sirve como herramienta para mantener relaciones de equivalencia dinámicas. Aunque tiene una implementación muy eficiente (¡y sencilla!), el análisis es complicado.

Las aplicaciones incluyen un algoritmo de árbol abarcante mínimo, que se verá en la sección 8.4, al igual que algunos problemas mencionados al final de esta sección.

6.6.1 Relaciones de equivalencia dinámicas

Una *relación de equivalencia R* sobre un conjunto *S* es una relación binaria sobre *S* que es reflexiva, simétrica y transitiva (sección 1.3.1). Es decir, para todo *s*, *t* y *u* en *S*, la relación satisface estas propiedades: *s Rs*; si *s Rt*, entonces *t Rs*; y si *s Rt* y *t Ru*, entonces *s Ru*. La clase de equivalencia de un elemento *s* en *S* es el subconjunto de *S* que contiene todos los elementos equivalentes a *s*. Las clases de equivalencia forman una partición de *S*, es decir, son disjuntas y su unión es *S*. De aquí en adelante usaremos el símbolo "≡" para denotar una relación de equivalencia.

El problema que estudiaremos en esta sección consiste en representar, modificar y contestar ciertas preguntas acerca de, una relación de equivalencia que cambia durante un cómputo. En un principio, la relación de equivalencia es la relación de igualdad, es decir, cada elemento está solo en un conjunto. El problema consiste en procesar una sucesión de instrucciones de los dos tipos siguientes, donde s_i y s_j son elementos de *S*:

1. ¿ES $s_i \equiv s_j$?
2. HACER $s_i \equiv s_j$ (donde $s_i \equiv s_j$ no es verdad todavía).

La pregunta 1 se contesta "sí" o "no". La respuesta correcta depende de las instrucciones del segundo tipo que se hayan recibido ya; la respuesta es afirmativa si y sólo si ya apareció la instrucción "HACER $s_i \equiv s_j$" o si se puede deducir que $s_i \equiv s_j$ aplicando las propiedades reflexiva, simétrica y transitiva a pares que se hicieron explícitamente equivalentes con el segundo tipo de instrucción. La respuesta a la segunda instrucción, HACER, consiste en modificar la estructura de datos que representa la relación de equivalencia de modo que instrucciones posteriores del primer tipo se contesten correctamente.

Consideremos el ejemplo siguiente en el que $S = \{1, 2, 3, 4, 5\}$. La sucesión de instrucciones aparece en la columna de la izquierda. La columna de la derecha muestra la respuesta, que puede ser afirmativa o negativa, o bien el conjunto de clases de equivalencia para la relación definido en ese momento.

Clases de equivalencia al comenzar: {1}, {2}, {3}, {4}, {5}

1.	¿ES $2 \equiv 4$?	No
2.	¿ES $3 \equiv 5$?	No
3.	HACER $3 \equiv 5$.	{1}, {2}, {3, 5}, {4}
4.	HACER $2 \equiv 5$.	{1}, {2, 3, 5}, {4}
5.	¿ES $2 \equiv 3$?	Sí
6.	HACER $4 \equiv 1$.	{1, 4}, {2, 3, 5}
7.	¿ES $2 \equiv 4$?	No

6.6.2 Algunas implementaciones obvias

Para comparar diversas estrategias de implementación, contaremos las operaciones de diferentes tipos efectuadas con cada estrategia para procesar una sucesión de m instrucciones HACER y/o ES sobre un conjunto S de n elementos. Comenzaremos por examinar dos estructuras de datos relativamente obvias para representar la relación: matrices y arreglos.

Una representación matricial de una relación de equivalencia requiere n^2 celdas (o aproximadamente $n^2/2$ si se usa la simetría). En el caso de una instrucción ES sólo es necesario examinar un elemento; en cambio, una instrucción HACER requeriría copiar varias filas. Una sucesión de m instrucciones HACER (o sea, una sucesión de peor caso de m instrucciones HACER y ES) requeriría por lo menos mn operaciones.

La cantidad de espacio empleada se puede reducir a n empleando un arreglo, digamos `claseEq`, en el que `claseEq[i]` es un rótulo o nombre para la clase de equivalencia que contiene a s_i. Una instrucción ¿ES $s_i \equiv s_j$? requiere acceder a `claseEq[i]` y `claseEq[j]` y compararlos. Una instrucción HACER $s_i \equiv s_j$ requiere examinar cada uno de los elementos para ver si es igual a `claseEq[i]` y, si lo es, asignarle `claseEq[j]`. Una vez más, con una sucesión de m instrucciones HACER (o sea, una sucesión de peor caso), se efectuarán por lo menos mn operaciones.

Ambos métodos tienen aspectos ineficientes: el copiado en el primero y la búsqueda (de elementos en `claseEq[i]`) en el segundo. Otras soluciones mejores usan ligas para evitar el trabajo extra.

6.6.3 Programas Unión-Hallar

El efecto de una instrucción HACER es formar la unión de dos subconjuntos de S. Una instrucción ES se puede contestar fácilmente si tenemos alguna forma de averiguar en qué conjunto está un elemento dado. El tipo de datos abstracto Unión-Hallar (sección 2.5.2) ofrece precisamente esas operaciones. En un principio se ejecuta `hacerConjunto` con cada elemento de S para crear n conjuntos de un solo elemento. Se usarán las operaciones `hallar` y `union` como sigue para implementar las instrucciones de equivalencia:

¿ES $s_1 \equiv s_j$?	HACER $s_1 \equiv s_j$
$t =$ `hallar`(s_i);	$t =$ `hallar`(s_i);
$u =$ `hallar`(s_j);	$u =$ `hallar`(s_j);
¿$(t == u)$?	`union`(t, u)

Usaremos `crear(n)` como abreviatura de

```
crear(0), hacerConjunto(1), hacerConjunto(2), ..., hacerConjunto(n).
```

Esto supone que $S = \{1, \ldots, n\}$. El resultado es una colección de conjuntos, cada uno de los cuales contiene un solo elemento, i, $1 \leq i \leq n$. Si es preciso agregar elementos uno por uno durante el programa, en vez de agregarse todos al principio, suponemos que se ejecuta `hacerConjunto` con ellos en sucesión numérica, sin huecos: `hacerConjunto(1)`, `hacerConjunto(2)`, ..., `hacerConjunto(k)`. Si ésta no es la numeración natural de los elementos, se puede usar un TDA Diccionario (sección 2.5.3) para la traducción.

Así pues, dirigiremos ahora nuestra atención a los operadores `hacerConjunto`, `union` y `hallar` y a una estructura de datos específica en la que se pueden implementar fácilmente. Representaremos cada clase de equivalencia, o subconjunto, con un árbol adentro. Recordemos que el tipo de datos abstracto Árbol adentro proporciona las operaciones siguientes:

`crearNodo`	construye un árbol de un solo nodo
`hacerPadre`	cambia el padre de un nodo
`ponerDatosNodo`	asigna un valor de datos entero al nodo
`esRaiz`	devuelve **`true`** si el nodo no tiene padre
`padre`	devuelve el padre del nodo
`datosNodo`	devuelve el valor de datos

Cada raíz se usará como rótulo o identificador de su árbol. La instrucción `r = hallar(v)` halla y asigna a *r* la raíz del árbol que contiene *v*. Los parámetros de `union` deben ser raíces; `union(t, u)` une los árboles cuyas raíces son *t* y *u* ($t \neq u$).

El TDA Árbol Adentro facilita la implementación de `union` y `hallar`. Para combinar las raíces *t* y *u*, donde *u* es la raíz del árbol adentro resultante, como requiere `union`, basta con ejecutar la operación de árbol adentro `hacerPadre`(*t*, *u*). Para hallar la raíz de un nodo usamos la función `padre` repetidamente hasta encontrar el antepasado para el cual `esRaiz` da **`true`**. La implementación de `crear` y `hacerConjunto` también es fácil utilizando la operación de árbol adentro `crearNodo`.

Si los nodos de un árbol adentro están numerados $1, \ldots, n$, donde $n = |S|$, podremos implementar el árbol adentro con unos cuantos arreglos de $n + 1$ elementos cada uno. Puesto que esto es lo que suele hacerse en la práctica, y con el fin de concentrarnos en los puntos fundamentales, adoptaremos este supuesto durante el resto de la sección. Podemos "desabstraer" el árbol adentro y simplemente acceder al elemento de arreglo `padre[i]` en lugar de invocar `padre(i)` como función de acceso o `hacerPadre(i)` como procedimiento de manipulación. Adoptaremos la convención de que un valor de `padre` de -1 denota que ese nodo de árbol adentro es una raíz, así que no necesitaremos un arreglo para `esRaiz`. Otro arreglo puede contener `datosNodo`, pero este nombre es demasiado general para la aplicación que nos ocupa, así que le daremos el nombre más específico de `peso`, previendo el método de unión ponderada que describiremos a continuación. Se puede usar doblado de arreglos (sección 6.2) si el número de elementos no se conoce con antelación.

El uso de arreglos simplifica el código, pero para entender la lógica de los algoritmos lo mejor es tener en mente la estructura de árbol adentro subyacente e interpretar los accesos a arreglos en términos de las operaciones de árbol. Esta implementación de árboles adentro mediante arreglos también se usa en varios otros algoritmos, así que vale la pena recordarla.

Una operación `crear(n)` (considerada como *n* operaciones `hacerConjunto`) seguida de una sucesión de *m* operaciones `union` y/o `hallar` intercaladas en cualquier orden se considerará como una entrada, o programa *Unión-Hallar*, de longitud *m*. Es decir, las instrucciones `hacerConjunto` iniciales no se cuentan en la longitud del programa. Para simplificar la explicación, supondremos que no vuelve a usarse `hacerConjunto` después del `crear` inicial. El análisis llega a las mismas concluisones generales si se vuelve a usar `hacerConjunto` después (véase el ejercicio 6.31).

Usaremos el número de accesos al arreglo `padre` como medida del trabajo realizado; cada acceso es una *consulta* o bien una *asignación*, y supondremos que cada uno tarda un tiempo $O(1)$. (Se hará evidente que el número total de operaciones es proporcional al número de accesos a `padre`.) Cada `hacerConjunto` o `union` efectúa una asignación a `padre`, cada `hallar(i)` efectúa $d + 1$ consultas de `padre`, donde *d* es la profundidad del nodo *i* en su árbol. Llamaremos colectivamente a las asignaciones y consultas de `padre` operaciones de *liga*.

1. *Unión* (1, 2)

2. *Unión* (2, 3)

⋮

$n-1$. *Unión* ($n-1$, n)

n. *Hallar* (1)

⋮

m. *Hallar* (1)

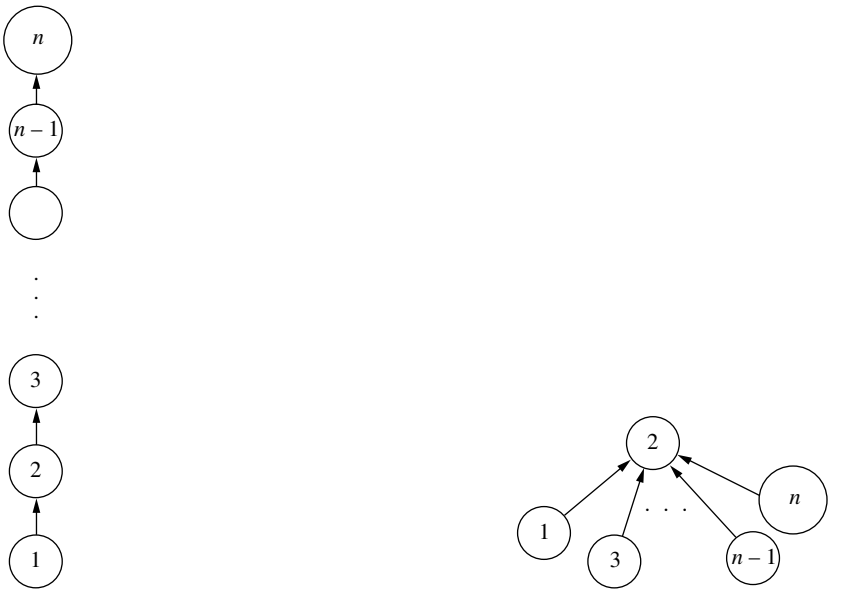
Figura 6.19 Programa *Unión-Hallar P* con $S = \{1, \ldots, n\}$ y que consiste en $n - 1$ uniones, seguidas de $m - n + 1$ búsquedas.

El programa de la figura 6.19 construye el árbol que se muestra en la figura 6.20(a) y efectúa $n + n - 1 + (m - n + 1)n$ operaciones de liga en total. Esto pone de manifiesto que, si se emplean estos métodos, el tiempo de peor caso para un programa *Unión-Hallar* está en $\Omega(mn)$. (Estamos suponiendo que $m > 0$; de lo contrario deberíamos escribir $\Omega(mn + n)$.) No es difícil demostrar que ningún programa semejante efectúa más de $mn + n$ operaciones de liga, de modo que el peor caso está en $\Theta(mn)$. En general, esto no es mejor que los métodos que describimos antes. Mejoraremos la implementación de las instrucciones `union` y `hallar`.

6.6.4 Unión ponderada

El costo del programa de la figura 6.19 es elevado porque el árbol que construyen las operaciones `union`, figura 6.20(a), es muy alto. Podríamos reducir su altura con una implementación más cuidadosa de `union` encaminada a producir árboles cortos. Sea `unionP` (por "unión ponderada") la estrategia que hace que el árbol que tiene menos nodos sea un subárbol de la raíz del otro árbol (y, digamos, que hace que el primer árbol sea un subárbol del segundo si ambos árboles tienen el mismo número de nodos). (El ejercicio 6.22 examina la posibilidad de usar la altura en lugar del número de nodos como "peso" de cada árbol.) Para distinguir entre las dos implementaciones de la operación `union`, llamaremos a la primera `unionNoP`, por "unión no ponderada". En el caso de `unionP`, el número de nodos de cada árbol se almacena en el arreglo `peso` (que corresponde a `datosNodo` en términos del TDA). En realidad, el valor sólo se necesita en la raíz. `unionP` debe comparar el número de nodos, calcular el tamaño del nuevo árbol, y efectuar asignaciones a `padre` y `peso`. El costo de una operación `unionP` sigue siendo una constante pequeña, que incluye una operación de liga. Si ahora volvemos al programa de la figura 6.19 (llamémoslo *P*) para ver qué tanto trabajo requiere si usamos `unionP`, nos encontraremos con que *P* ha dejado de ser un programa válido porque los parámetros de `union` en las instrucciones 3 a $n - 1$ no son todos raíces. Podríamos expandir *P* para dar el programa *P′* sustituyendo cada instrucción de la forma `union(i,j)` por las tres instrucciones

```
t = hallar(i);
u = hallar(j);
union(t, u);
```



(a) Árbol para P, empleando unión no ponderada

(b) Árbol para P', empleando unión ponderada

Figura 6.20 Árboles que se obtienen empleando unión no ponderada y unión ponderada

Así pues, si usamos `unionP`, ¡P' sólo requerirá $2m + 2n - 1$ operaciones de liga! La figura 6.20 muestra los árboles que se construyen para P y P' empleando `unionNoP` y `unionP`, respectivamente. No podemos concluir que `unionP` haga posibles implementaciones en tiempo lineal en todos los casos; P' no es un programa de peor caso para `unionP`. El lema siguiente nos ayuda a obtener una cota superior para el peor caso.

Lema 6.6 Si `union(t, u)` se implementa con `unionP` —es decir, de modo que el árbol cuya raíz es u se conecte como subárbol de t si y sólo si el número de nodos del árbol con raíz en u es menor, en caso contrario el árbol con raíz en t se conecte como subárbol de u— entonces, después de cualquier sucesión de instrucciones `union`, cualquier árbol que tenga k nodos tendrá una altura máxima de $\lfloor \lg k \rfloor$.

Demostración La demostración es por inducción con k. El caso base es $k = 1$; un árbol con un nodo tiene altura 0, que es $\lfloor \lg 1 \rfloor$. Supóngase ahora que $k > 1$ y que cualquier árbol construido mediante una sucesión de instrucciones `union` y que contiene m nodos, para $m < k$, tiene una altura máxima de $\lfloor \lg m \rfloor$. Consideremos el árbol T de la figura 6.21 que tiene k nodos, altura h y se construyó a partir de los árboles T_1 y T_2 mediante una instrucción `union`. Supóngase, como se indica en la figura, que u, la raíz de T_2, se conectó a t, la raíz de T_1. Sean k_1 y h_1 el número de nodos y la altura de T_1, respectivamente, y k_2 y h_2 los valores correspondientes para T_2. Por la hipótesis inductiva, $h_1 \leq \lfloor \lg k_1 \rfloor$ y $h_2 \leq \lfloor \lg k_2 \rfloor$. La altura del nuevo árbol es $h = \text{máx}(h_1, h_2 + 1)$. Es evidente que $h_1 \leq \lfloor \lg k \rfloor$. Puesto que $k_2 \leq k/2$, $h_2 \leq \lfloor \lg k \rfloor - 1$, y $h_2 + 1 \leq \lfloor \lg k \rfloor$. Por tanto, en ambos casos $h \leq \lfloor \lg k \rfloor$. □

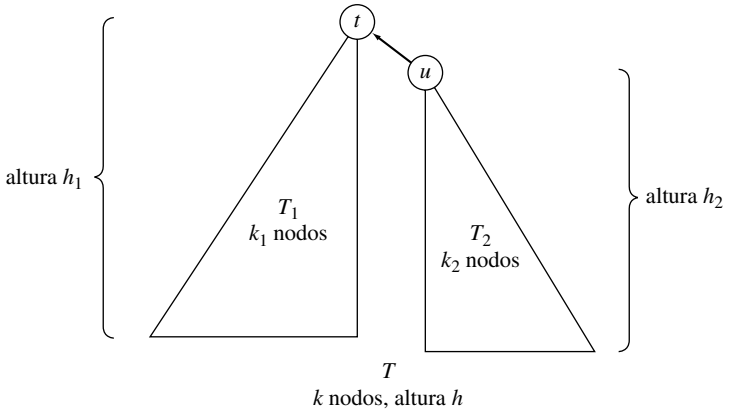


Figura 6.21 Ejemplo para la demostración del lema 6.6

Teorema 6.7 Un programa *Unión-Hallar* de tamaño m, ejecutado con un conjunto de n elementos, efectúa $\Theta(n + m \log n)$ operaciones de liga en el peor caso si se usan `unionP` y la instrucción `hallar` sencilla.

Demostración Con n elementos, se pueden ejecutar cuando más $n - 1$ instrucciones `unionP`, con lo que se construye un árbol con un máximo de n nodos. Por el lema, cada árbol tiene una altura máxima de $\lfloor \lg n \rfloor$, así que el costo de cada `hallar` es cuando más $\lfloor \lg n \rfloor + 1$. Cada `unionP` efectúa una operación de liga, así que el costo de m operaciones `hallar` es una cota superior para el costo de cualquier combinación de m operaciones `unionP` o `hallar`. El número total de operaciones de liga es, por tanto, menor que $m(\lfloor \lg n \rfloor + 1)$, que está en $O(n + m \log n)$. □

Demostrar que, por ejemplo, es posible construir programas que requieren $\Omega(n + m \log n)$ pasos, se deja para el ejercicio 6.23.

Los algoritmos para `unionP` (y también para `crear` y `hacerConjunto`) son muy fáciles de escribir; los dejaremos como ejercicios.

6.6.5 Compresión de caminos

La implementación de la operación `hallar` también se puede modificar para acelerar un programa *Unión-Hallar* mediante un proceso llamado *compresión de caminos*. Dado el parámetro v, `hallarCC` (por "hallar con compresión") sigue a los padres del nodo de v hasta la raíz y luego restablece los padres en todos los nodos del camino recién recorrido de modo que todos apunten a la raíz. Véase la figura 6.22.

El efecto de `hallarCC` se ilustra en la figura 6.23. La omisión de las líneas 6 y 7 da el procedimiento para el `hallar` sencillo.

Hay una operación de liga en la línea 1 (efectuada por `hallar` y `hallarCC`) y una en la línea 7 (efectuada sólo por `hallarCC`). Por tanto, la función `hallarCC` realiza el doble de operaciones de liga que la `hallar` sencilla para un nodo específico de un árbol dado, pero el uso de

```
int hallarCC(int v)
    int raiz;
1.  int antiguoPadre = padre[v];
2.  if (antiguoPadre == −1)  // v es una raíz
3.     raiz = v;
4.  else
5.     raiz = hallarCC(antiguoPadre);
6.     if (antiguoPadre ≠ raiz)  // Este enunciado if
7.        padre[v] = raiz;  // efectúa compresión de camino.
8.  return raiz;
```

Figura 6.22 Procedimiento para `hallarCC`

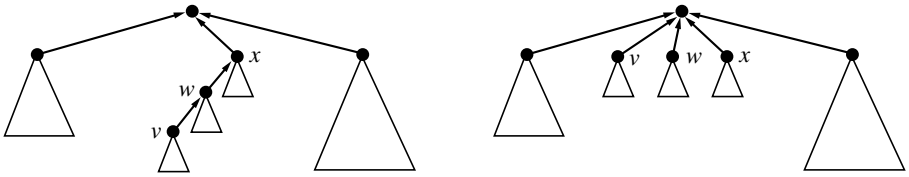


Figura 6.23 Hallar con compresión de caminos (`hallarCC`)

`hallarCC` mantiene los árboles muy cortos, por lo que el trabajo total se reduce. Se puede demostrar (véanse las Notas y referencias al final del capítulo) que, si se usan `hallarCC` y `unionNoP` (la unión *no ponderada*), el tiempo de ejecución de peor caso de programas con longitud m está en $O(n + m \log n)$. Los ejercicios 6.25 a 6.29 muestran que efectivamente existe un programa que requiere $\Theta(n + m \log n)$ pasos. Así pues, el uso de la implementación mejorada de `union` o bien la implementación mejorada de `hallar` reduce la complejidad de peor caso de un programa, de $\Theta(n + mn)$ a $\Theta(n + m \log n)$. El siguiente paso consiste en combinar las dos mejoras, con la esperanza de lograr una reducción aún mayor.

Compatibilidad de `unionP` y `hallarCC`

¿Son compatibles `unionP` y `hallarCC`?, `hallarCC` modifica la estructura del árbol sobre el que actúa pero no altera el número de nodos de ese árbol, aunque sí podría cambiar su altura. Recordemos que podría haber parecido más natural que `unionP` comparara las alturas de los árboles que está uniendo en lugar del número de nodos de cada uno, pues lo que se busca es que los árboles no sean muy altos. Sería difícil actualizar la altura de un árbol correctamente una vez que `hallarCC` lo ha modificado. Se usó el número de nodos como peso específicamente para que `unionP` y `hallarCC` sean compatibles.

*6.6.6 Análisis de `unionP` y `hallarCC`

Ahora deduciremos una cota superior para el número de operaciones de liga que efectúa un programa *Unión-Hallar* que usa `unionP` y `hallarCC`, aplicando la técnica de análisis amortizado que presentamos en la sección 6.3. En esta explicación, P es un programa *Unión-Hallar* de longitud m que opera sobre el conjunto de elementos $S = \{1, \ldots, n\}$. Necesitamos varias definiciones y lemas para obtener el resultado deseado, que es el teorema 6.13.

Definición 6.8 Bosque F, altura de nodo, rango

Para un programa *Unión-Hallar* específico, P, sea F el bosque construido por la sucesión de instrucciones `union` de P, suponiendo que se usa `unionP` y que se hace caso omiso de las instrucciones `hallar`. La *altura* de un nodo v en cualquier árbol es la altura del subárbol cuya raíz es v. La altura de un nodo v en F se define como el *rango de* v. ■

Deduciremos unas cuantas propiedades de F.

Lema 6.8 En el conjunto S hay cuando más $n/2^r$ nodos con rango r, para $r \geq 0$.

Demostración Del lema 6.6 se sigue que cualquier árbol de altura r construido por una sucesión de operaciones `unionP` tiene por lo menos 2^r nodos. Cada uno de los subárboles de F (es decir, un nodo y todos sus descendientes) fue en algún momento un árbol individual, así que cualquier subárbol de F cuya raíz es un nodo de rango r tiene por lo menos 2^r nodos. Puesto que los subárboles cuya raíz tiene rango r son disjuntos, no puede haber más de $n/2^r$ de esos subárboles. □

Lema 6.9 Ningún nodo de S tiene rango mayor que $\lfloor \lg n \rfloor$.

Demostración Usamos el lema 6.6 y el hecho de que S sólo tiene n nodos. □

Los lemas 6.8 y 6.9 describen propiedades del bosque F construido por las instrucciones `union` de un programa *Unión-Hallar*, haciendo caso omiso de las instrucciones `hallar`. Si las instrucciones `hallar` incluidas en P se ejecutan empleando `hallarCC`, el resultado será un bosque distinto y la altura de los diversos nodos será diferente de su rango, que se basa en F.

Lema 6.10 En cualquier momento durante la ejecución de un programa *Unión-Hallar* P, los rangos de los nodos que están en un camino que va de una hoja hasta una raíz de un árbol forman una sucesión estrictamente creciente. Cuando una operación `hallarCC` cambia el padre de un nodo, el nuevo padre tiene un rango más alto que el antiguo padre de ese nodo.

Demostración Es indudable que en F los rangos forman una sucesión creciente en un camino de una hoja a la raíz. Si, durante la ejecución de P, un nodo v se convierte en hijo de un nodo w, v deberá ser un descendiente de w en F, así que el rango de v es menor que el de w. Si v se convierte en un hijo de w debido a una operación `hallarCC`, ello querrá decir que w era un antepasado del padre anterior de v, de ahí se sigue la segunda afirmación del lema. □

i	0	1	2	3	4	5	6	⋯	16	17	⋯	65536	65537
$H(i)$	1	2	4	16	56536	2^{65536}	??						
lg*(i)		0	1	2	2	3	3	⋯	3	4	⋯	4	5

Tabla 6.1 Las funciones H y lg*

En el teorema 6.13 estableceremos una cota superior de $O(n \lg^*(n))$ para el tiempo de ejecución de un programa *Unión-Hallar* que usa `unionP` y `hallarCC`, donde lg* es una función que crece con extrema lentitud.

Definición 6.9 Log-asterisco

Para definir lg* primero definimos la función H como sigue:

$$\begin{aligned} H(0) &= 1, \\ H(i) &= 2^{H(i-1)} \qquad \text{para } i > 0. \end{aligned}$$

Por ejemplo,

$$H(5) = 2^{2^{2^{2^2}}}.$$

lg*(j) se define para $j \geq 1$ como el i más pequeño tal que $H(i) \geq j$; es decir, informalmente, lg*(j) es el número de doses que es preciso "apilar" para alcanzar o exceder a j. ■

Por la definición, es obvio que lg*(n) está en $o(\log^{(p)} n)$ para cualquier constante $p \geq 0$. (Usamos la convención de que $\lg^{(0)} n = n$.) En la tabla 6.1 se muestran algunos valores de H y lg*. Para cualquier entrada concebible que pudiera llegar a usarse, lg* $n \leq 5$.

Ahora dividiremos los nodos de S en grupos, según su rango. El esquema contable del costo amortizado se basará en estos grupos de nodos.

Definición 6.10 Grupos de nodos

Definimos s_i para $i \geq 0$ como el conjunto de nodos $v \in S$ tales que lg*(1 + rango de v) = i. La relación entre rangos y grupos para valores pequeños está dada por la tabla siguiente:

r (rango)	0	1	2−3	4−15	16−65535	$65536-(2^{65536}-1)$
i (grupo)	0	1	2	3	4	5

■

Lema 6.11 El número de grupos de nodos distintos para S es cuando más lg*(n + 1).

Demostración El rango de cualquier nodo es cuando más $\lfloor \lg n \rfloor$. El índice de grupo máximo es

$$\lg^*(1 + \lfloor \lg n \rfloor) = \lg^*(\lceil \lg(n+1) \rceil) = \lg^*(n+1) - 1,$$

y el índice de grupo mínimo es 0. □

Ya estamos en condiciones de definir costos contables, de los que deduciremos costos amortizados empleando la ecuación (6.1), que repetimos aquí para comodidad del lector:

$$\textit{costo amortizado} = \textit{costo real} + \textit{costo contable}.$$

Recordemos que el programa consiste en n invocaciones de `hacerConjunto`, seguidas de una combinación arbitraria de m operaciones `union` y `hallar`, con la salvedad de que no hay más de $n - 1$ uniones. El ejercicio 6.31 considera el caso en el que las invocaciones a `hacerConjunto` podrían estar intercaladas en todo el programa y no se conoce n con antelación.

Definición 6.11 Costos de `unionP` y `hallarCC`

Los costos de las operaciones del TDA Unión-Hallar son los siguientes. El costo unitario es "operaciones de liga" (asignaciones a padres y consultas).

1. El costo contable de `hacerConjunto` es $4 \lg^*(n + 1)$. Pensemos en estos costos contables positivos como depósitos en una cuenta de ahorros. El costo real es 1 (el de asignar -1 al padre). El costo amortizado es $1 + 4 \lg^*(n + 1)$.
2. El costo contable de `unionP` es 0. El costo real es 1. El costo amortizado es 1.
3. El costo contable de `hallarCC` es el más complicado. Supóngase que, en el momento en que se invoca `hallarCC(v)` (pero no recursivamente desde otra invocación de `hallarCC`), el camino desde v hasta la raíz de su árbol adentro está dado por la sucesión $v = w_0, w_1, \ldots, w_k$, donde w_k es la raíz. Usamos la convención de que $k = 0$ si v es una raíz. Si k es 0 o 1, el costo contable es 0 (y ningún padre se modifica).

   Para $k \geq 2$, el costo contable es -2 por cada par (w_{i-1}, w_i) tal que $1 \leq i \leq k - 1$ y los grupos de nodos de w_{i-1} y w_i, según la definición 6.10, son los mismos. Cada uno de estos cargos de -2 es un *retiro por* w_{i-1}. Cabe señalar que los rangos de w_i aumentan al hacerlo i, así que los grupos de nodos forman una sucesión no decreciente.

   El costo real de `hallarCC` es $2k$, porque la raíz y el hijo de la raíz efectúan una consulta, pero ninguna asignación a un padre. Por tanto, el *costo amortizado* es de 2 más 2 veces el número de casos en que w_{i-1} está en un grupo de nodos *diferente* del de w_i, para $1 \leq i \leq k - 1$. Por el lema 6.11, el costo amortizado de cualquier `hallarCC` es cuando más $2 \lg^*(n + 1)$. ■

Aunque el costo de peor caso de una `hallarCC` podría ser $2 \lg n$, el esquema de amortización ha repartido algo del costo a la operación `crear` inicial. ¿Equivale esto a quitarle a Pedro para pagarle a Pablo?, no precisamente. Observemos que los cargos contables en los que incurre `hacerConjunto` dependen únicamente de n, el número de elementos del conjunto. En cambio, el número de operaciones `hallarCC` es de por lo menos $m - n + 1$, que puede ser arbitrariamente mayor que n. No obstante, el costo amortizado por cada `hallarCC` es de únicamente $2 \lg^*(n + 1)$, un ahorro considerable respecto a $2 \lg n$. Todo esto es muy bonito, pero falta ver si podemos "costearlo": que la "cuenta de ahorros" establecida por `crear` nunca se sobregirará.

Lema 6.12 El sistema de costos contables de la definición 6.11 produce un esquema de costo amortizado válido en el sentido de que la suma de los costos contables nunca es negativa.

Demostración Las operaciones `hacerConjunto` iniciales hacen que la suma de costos contables ascienda a $4n \lg^*(n+1)$. Bastará demostrar que la suma de los cargos negativos en los que incurren las operaciones `hallarCC` no exceden ese total.

Cada cargo negativo se identifica como un retiro para algún nodo, digamos w. Esto ocurre si w está en el camino recorrido por una `hallarCC` y pertenece al mismo grupo de nodos que su padre, y su padre no es una raíz. Sea i ese grupo de nodos. Entonces, esta `hallarCC` asignará un nuevo padre a w, por el lema 6.10 el nuevo padre tendrá un rango más alto que el antiguo padre. Una vez que se haya asignado a w un padre nuevo en un *grupo de nodos* más alto, ya no estará asociado a más retiros. Por tanto, w no puede estar asociado a más retiros que nodos hay en su grupo de nodos. El número de rangos del grupo i es menor que $H(i)$, por la definición 6.9, y ésta es una cota superior del número de retiros para w.

El número de retiros para todos los $w \in S$ es cuando más

$$\sum_{i=0}^{\lg^*(n+1)-1} H(i) \text{ (número de nodos del grupo } i). \tag{6.2}$$

Por el lema 6.8, no hay más de $n/2^r$ nodos con rango r, así que el número de nodos en el grupo i es

$$\sum_{r=H(i-1)}^{H(i)-1} \frac{n}{2^r} \leq \frac{n}{2^{H(i-1)}} \sum_{j=0}^{\infty} \frac{1}{2^j} = \frac{2n}{2^{H(i-1)}} = \frac{2n}{H(i)}.$$

Por tanto, la sumatoria de la ecuación (6.2) está acotada por arriba de acuerdo con

$$\sum_{i=0}^{\lg^*(n+1)-1} H(i)\left(\frac{2n}{H(i)}\right) = 2n \lg^*(n+1).$$

Cada retiro es -2, así que la suma de los retiros no puede exceder $4n \lg^*(n+1)$. □

Teorema 6.13 El número de operaciones de liga efectuadas por un programa *Unión-Hallar* implementado con `unionP` y `hallarCC`, de longitud m y ejecutado con un conjunto de n elementos, está en $O((n+m)\lg^*(n))$ en el peor caso.

Demostración El esquema de amortización definido en la definición 6.11 da costos amortizados de cuando más $1 + 4 \lg^*(n+1)$ por cada operación Unión-Hallar. Hay $n + m$ operaciones, incluidas las `hacerConjunto`. Una cota superior para el costo total amortizado es $(n+m)(1 + 4 \lg^*(n+1))$. Por el lema 6.12, la suma de los costos reales nunca excede la suma de los costos amortizados, así que la cota superior también es válida para los costos reales. □

Puesto que $\lg^* n$ crece con gran lentitud y las estimaciones hechas en la demostración del teorema son relativamente holgadas, es natural preguntarse si podríamos demostrar un teorema más categórico, es decir, que el tiempo de ejecución de los programas *Unión-Hallar* de longitud m ejecutados con un conjunto de n elementos e implementado con `unionP` y `hallarCC` está en $\Theta(n+m)$. Se ha demostrado que no es así (véanse las Notas y referencias al final del capítulo). Para cualquier constante c, hay programas de longitud m ejecutados con conjuntos de tamaño n que requieren más de cm operaciones utilizando éstas (y varias otras) técnicas. No obstante, véase el ejercicio 6.30.

Es imposible asegurar si existen o no otras técnicas que implementen programas *Unión-Hallar* en tiempo lineal. No obstante, como pone de manifiesto el teorema 6.13, el uso de `hallarCC` y `unionP` produce una implementación muy eficiente de los programas *Unión-Hallar*. Supondremos esta implementación al tratar aplicaciones posteriores.

Programas de equivalencia

Comenzamos con un intento de hallar una buena forma de representar una relación de equivalencia dinámica de modo tal que las instrucciones de las formas HACER $s_i \equiv s_j$ y ¿ES $s_i \equiv s_j$? se pudieran manejar con eficiencia. Definimos un programa de equivalencia de longitud m como una sucesión de m instrucciones de ese tipo intercaladas en cualquier orden. Puesto que, como observamos antes, cada instrucción HACER o ES se puede implementar con tres instrucciones del conjunto `unionP`, `hallarCC`, prueba de igualdad, un programa de equivalencia de longitud m que se ejecuta con un conjunto de n elementos se puede implementar en tiempo $O((m + n) \lg^* n)$.

6.6.7 Aplicaciones

Una de las aplicaciones más conocidas de un programa de equivalencia es el algoritmo de árbol abarcante mínimo de Kruskal. Analizaremos este algoritmo en la sección 8.4, después de introducir material necesario acerca de grafos. Aquí describiremos brevemente otras aplicaciones. En las Notas y referencias al final del capítulo se menciona bibliografía sobre estas aplicaciones. En general, un programa de equivalencia es lo indicado cuando es preciso procesar información conforme se recibe, descubre o calcula. Esto se denomina operación *en línea*.

Los operadores `union` y `hallar` pueden servir para implementar una sucesión de otros dos tipos de instrucciones que actúan sobre la misma especie de estructuras de árbol: `ligar(r, v)`, que hace que el árbol cuya raíz es r sea un subárbol de v, y `profundidad(v)`, que determina la profundidad actual de v. Es posible implementar una sucesión de tales instrucciones en tiempo $O(n \lg^*(n))$.

El estudio de programas de equivalencia se justifica por el problema de procesar declaraciones de **`equivalencia`** en Fortran y otros lenguajes de programación. Una declaración de **`equivalencia`** indica que dos o más variables o elementos de arreglo deben compartir las mismas posiciones de almacenamiento. El problema consiste en asignar correctamente direcciones de almacenamiento a todas las variables y arreglos. La declaración

```
equivalencia (A,B(3)),(B(4),C(2)),(X,Y,Z),(J(1),K),(B(1),X),(J(4),L,M)
```

indica que `A` y `B(3)` comparten la misma posición, `B(4)` y `C(2)` comparten la misma posición, etc. (En Fortran se usan paréntesis, no corchetes, para los índices de arreglos.) La organización de memoria completa indicada por este enunciado **`equivalencia`** se muestra en la figura 6.24, que supone por sencillez que cada uno de los arreglos tiene cinco elementos.

Si no hubiera arreglos, el problema de procesar declaraciones de **`equivalencia`** (tan pronto como aparecen en el programa fuente) sería básicamente el de procesar un programa de equivalencia. La inclusión de arreglos requiere un poco de contabilidad adicional e introduce la posibilidad de que una declaración no sea aceptable. Por ejemplo,

```
equivalencia (A(1),B(1)),A(2),C(3)),(B(5),C(5))
```

no se permitiría porque los elementos de cada arreglo deben ocupar posiciones de memoria consecutivas.

$B(1)$ X Y Z	$B(2)$	A $B(3)$ $C(1)$	$B(4)$ $C(2)$	$B(5)$ $C(3)$	$C(4)$	$C(5)$	$J(1)$ K	$J(2)$	$J(3)$	$J(4)$ L M	$J(5)$

Figura 6.24 Organización de almacenamiento (con sintaxis de arreglos de Fortran) para una **equivalencia** `(A,B(3)),(B(4),C(2)), (X,Y,Z), (J(1),K), (B(1),X), (J(4),L,M)`.

Las operaciones `union` y `hallar` son sólo dos de muchas operaciones que pueden aplicarse a colecciones de subconjuntos. Otras son `insertar`, que inserta un miembro nuevo en un conjunto; `borrar`, que elimina un miembro de un conjunto; `min`, que halla el miembro más pequeño de un conjunto; `interseccion`, que produce un conjunto nuevo cuyos miembros son todos los elementos presentes en *los dos* conjuntos dados; y `miembro`, que indica si un elemento dado está o no en cierto conjunto. Se han estudiado técnicas y estructuras de datos para procesar eficientemente "programas" que consisten en sucesiones de dos o tres tipos de tales instrucciones. En algunos casos se pueden usar las técnicas de `union` y `hallar` para implementar programas de ese tipo y tamaño n en tiempo $O(n \lg^*(n))$.

6.7 Colas de prioridad con operación de decrementar clave

Recordemos que la principal función de acceso del TDA de cola de prioridad (sección 2.5.1) es `obtenerMejor`, donde "mejor" es el mínimo o el máximo. Las operaciones de un TDA de cola de prioridad *minimizante* completo son:

Constructor:	`crear`
Funciones de acceso:	`estaVacio, obtenerMin, obtenerPrioridad`
Procedimientos de manipulación:	`insertar, borrarMin, decrementarClave`

Se hacen las modificaciones apropiadas a los nombres en el caso de una cola de prioridad *maximizante*. También podría añadirse una operación `borrar`, que elimina una clave arbitraria.

Los árboles en orden parcial (definición 4.2) son una familia de estructuras de datos que se usa a menudo para implementar colas de prioridad. El "mejor" elemento está en la raíz del árbol en orden parcial, así que puede recuperarse en tiempo constante. Con el paso de los años, se han desarrollado varias implementaciones de los árboles en orden parcial. Es común encontrar la palabra "montón" (*heap*) en su nombre porque el inventor de la primera estructura para árboles en orden parcial la llamó "montón".

El montón binario, que se introdujo para Heapsort en la sección 4.8.1, permite implementar todos los procedimientos de manipulación en tiempo $O(\log n)$ cuando la cola de prioridad contiene n elementos. La fuerza impulsora para continuar las investigaciones fue que algunas aplicaciones usan la operación `decrementarClave` con mucha mayor frecuencia que cualquier otra, por lo que era deseable hacer más eficiente a esa operación sin elevar demasiado los demás costos. Se escogieron los bosques de apareamiento para incluirlos en esta sección porque son el más sencillo de muchos sistemas diseñados para hacer muy eficiente la operación `decrementarClave`. En las Notas y referencias al final del capítulo se mencionan fuentes que ofrecen alternativas más avanzadas.

En esta sección primero describiremos la forma de implementar la operación `decrementarClave` en un montón binario. Se necesita una estructura de datos auxiliar, la cual también puede hacer que una operación `borrar` con un elemento arbitrario sea eficiente. Luego describiremos la estrategia de bosque de apareamiento, que también usa una estructura auxiliar parecida, aunque su estructura de datos principal es un bosque general en orden parcial, no un árbol binario en orden parcial.

Recordemos que un montón binario es un árbol binario completo a la izquierda; es decir, todos los niveles del árbol están llenos con la posible excepción del nivel más bajo y todos los nodos de ese nivel se empacan a la izquierda sin huecos. Por consiguiente, los nodos se pueden almacenar en un arreglo con la raíz en la posición 1, y el árbol binario se puede recorrer siguiendo la regla de que los hijos del nodo que está en la posición k están en las posiciones $2k$ y $2k + 1$. El número de elementos, n, se almacena en una variable aparte.

6.7.1 La operación decrementar clave

No todas las aplicaciones de cola de prioridad necesitan las operaciones `decrementarClave` y `obtenerPrioridad`. Si se necesitan, la implementación se vuelve un poco más complicada. El nombre `decrementarClave` supone implícitamente un montón minimizante, lo cual es usual en problemas de optimización, en esta sección trabajaremos con montones minimizantes.

El problema es que la operación `decrementarClave` especifica un elemento que ya está *en algún lugar* de la cola de prioridad. La operación `obtenerPrioridad` sólo se necesita junto con `decrementarClave`; una vez resueltos los problemas de `decrementarClave`, `obtenerPrioridad` será fácil, por lo que no hablaremos más de ella. Si utilizamos un montón para la cola de prioridad, la rúbrica de `decrementarClave` sería similar a:

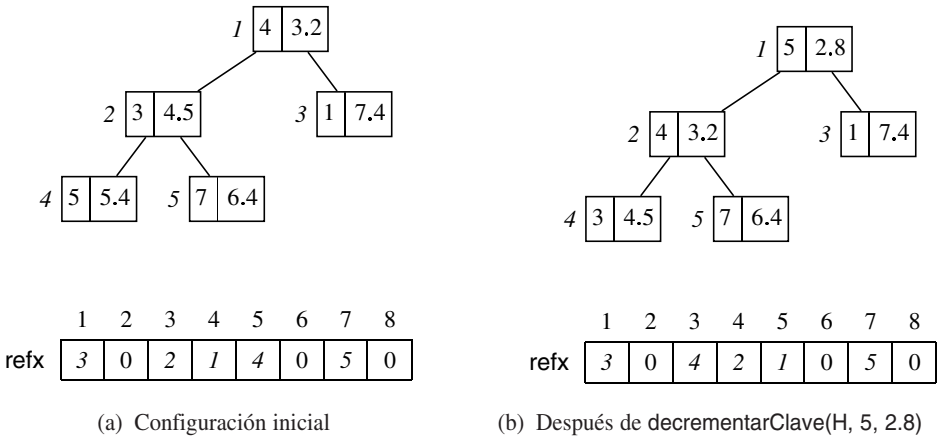
```
void decrementarClave(Clave[] H, int id, Clave K)
```

donde `id` es el identificador del elemento a modificar y K es el nuevo valor de clave (prioridad). La labor de la operación consiste en hallar ese elemento, decrementar su "clave", que puede verse como un costo, y ajustar su posición en la cola de prioridad según la nueva clave. Una vez que se ha localizado el elemento y se ha modificado su clave, es evidente que podremos usar `subirMonton` (algoritmo 4.10) para realizar el ajuste de posición (cambiando "$\leq$" a "$\geq$" en el código de la sección 4.8.6, donde estábamos usando un montón maximizante), porque el elemento se mueve hacia la raíz, si es que se mueve.

Sería muy ineficiente realizar una búsqueda por todo el montón para hallar el elemento con el identificador requerido, `id`. La técnica consiste en mantener una estructura de datos complementaria, organizada por identificadores, que indica en qué posición del montón está cada elemento. Si los identificadores son enteros dentro de un intervalo razonablemente compacto, que es el caso más común, la estructura de datos complementaria puede ser un arreglo. En general, la estructura de datos complementaria puede estar en un TDA Diccionario. Supondremos el caso más sencillo, que el identificador es un entero y llamaremos al arreglo complementario `refx`.

Ejemplo 6.13 Montón y arreglo `refx`

La figura 6.25 muestra un ejemplo pequeño de montón con arreglo complementario `refx` para acelerar la localización de elementos arbitrarios en el montón. Si `refx[id]` = 0, entonces el elemento `id` no está en el montón. Para ejecutar `decrementarClave(H, 5, 2.8)`, se consulta el



(a) Configuración inicial (b) Después de decrementarClave(H, 5, 2.8)

Figura 6.25 Montón *H* y arreglo complementario `refx`. El identificador de elemento y la clave se muestran dentro de los nodos; la clave es de punto flotante. Los índices de montón se muestran en cursivas fuera de los nodos.

arreglo `refx` para determinar que el elemento 5 está actualmente en la posición 4 del montón. (La operación `decrementarClave` se puede ejecutar con cualquier nodo del montón, no sólo las hojas.)

La clave del elemento 5 se cambia a 2.8. Todo el elemento (5, 2.8) se cambia a una posición temporal *K*. La posición 4 del montón está ahora vacía. Luego se invoca `subirMonton(H, 1, K, 4)`. El elemento *K* sube para restaurar la propiedad de árbol en orden parcial. Durante la operación, el elemento 3 pasa a la posición 4 del montón, el elemento 4 pasa a la posición 2 del montón y el elemento 5 pasa a la posición 1 del montón. El arreglo `refx` se actualiza conforme se efectúa cada traslado de elemento. Es preciso agregar código a `subirMonton` (y a todas las operaciones de montón que cambian de lugar elementos) para mantener actualizado a `refx`. ■

Puesto que los elementos nunca cambian de lugar dentro del arreglo `refx` (siempre se accede a ellos empleando su identificador como índice del arreglo), podría ser más eficiente o recomendable en algunos casos almacenar ahí las prioridades, no en el arreglo del montón.

6.7.2 Bosques de apareamiento

El objetivo principal de la implementación como Bosque de apareamiento del TDA Cola de Prioridad es hacer que la operación `decrementarClave` sea muy eficiente sin elevar demasiado los costos de las demás operaciones. En capítulos posteriores veremos algoritmos con los cuales `decrementarClave` se ejecuta muchas veces más que cualquiera de las otras operaciones de cola de prioridad. En esta sección nos concentraremos en las colas de prioridad *minimizantes* con el fin de simplificar la explicación. Las aplicaciones que necesitan la operación `decrementarClave` normalmente tratan de reducir al mínimo un costo de algún tipo, en lugar de aumentar algo al máximo.

La estrategia de Bosque de apareamiento utiliza la variante de una estructura de datos llamada *montón de apareamiento*. La implementación del Bosque de apareamiento es relativamente

sencilla, y ha tenido un buen desempeño en la práctica. No obstante, se sabe que no es óptima asintóticamente. Véanse las Notas y referencias al final del capítulo.

Un Bosque de apareamiento es una colección de árboles fuera generales que tienen la propiedad de árbol en orden parcial (definición 4.2); es decir, en cada camino desde la raíz de un árbol hasta una hoja, se llega a los nodos en orden de costo creciente, o de valor del campo de prioridad creciente. Las raíces de los árboles tienen la prioridad más baja de todos los nodos de su respectivo árbol. Sin embargo, no se conoce alguna relación de orden entre las raíces de los diversos árboles del bosque.

El Bosque de apareamiento en sí se puede representar como una lista ligada del tipo `ListaArboles`; sea el campo de ejemplar `bosque` de la clase `BosqueApareamiento` dicha lista. Los árboles y subárboles de este bosque son del tipo `Arbol`. Ya describimos los TDA `Arbol` y `ListaArboles` en la sección 2.3.4. Usaremos sus operaciones, que incluyen `construirArbol`, `raiz` e `hijos`, además de `cons`, `primero` y `resto`.

La esencia de los Bosques de apareamiento es el método para hallar el mínimo. En tanto haya dos o más árboles en el bosque, los árboles se organizarán en pares, como si fuera a efectuarse un torneo. La operación básica, llamada `aparearArbol`, toma dos árboles, `t1` y `t2`, compara sus raíces y combina ambos árboles en uno solo, con la raíz "ganadora" como raíz del árbol combinado. Puesto que estamos minimizando, el "ganador" es el nodo de menor prioridad. Se devuelve el árbol combinado.

```
Arbol aparearArbol(Arbol t1, Arbol t2) // BOSQUEJO
   Arbol nuevoArbol;
   if (raiz(t1).prioridad < raiz(t2).prioridad)
      nuevoArbol = construirArbol(raiz(t1), cons(t2, hijos(t1)));
   else
      nuevoArbol = construirArbol(raiz(t2), cons(t1, hijos(t2)));
   return nuevoArbol;
```

Observe el parecido con la operación de unión ponderada de la sección 6.6.4.

El bosque se mantiene como una lista de árboles. La operación `aparearBosque` ejecuta `aparearArbol` con cada par de árboles del bosque. Si en un principio el bosque tenía k árboles, `aparearBosque` reduce este número a $\lceil k/2 \rceil$ y devuelve la lista de árboles resultante.

```
aparearBosque(viejoBosque) // BOSQUEJO
   Suponer que viejoBosque = t1, t2, ..., tk.
   Aplicar aparearArbol a (t1, t2), (t3, t4), ..., y colocar los árboles resultantes en la lista
   nuevoBosque, de modo que el resultado de aparear (t1, t2) quede al final de la lista. Si k
   es impar, tk quedará al principio de nuevoBosque; de lo contrario, será el resultado de
   aparear tk-1 y tk.
   return nuevoBosque;
```

La función `obtenerMin` realiza rondas del torneo invocando repetidamente a `aparearBosque` hasta establecer un solo ganador (elemento mínimo).

```
obtenerMin(cp) // BOSQUEJO
   while (cp.bosque tenga más de un árbol)
     cp.bosque = aparearBosque(cp.bosque);
   min = campo de identificador de la raíz del único árbol restante.
   return min;
```

Este torneo es básicamente el mismo que se usó en el algoritmo para hallar los elementos máximo y segundo más grande de un conjunto en la sección 5.3.2, con la excepción de que aquí estamos buscando el mínimo y vamos combinando árboles conforme avanzamos, empleando la operación `aparearArbol`. En la figura 6.26 se muestra un ejemplo.

Como siempre, se necesitan $k - 1$ comparaciones de claves para hallar el mínimo de k elementos (k raíces de árboles en el bosque inicial). Puesto que k puede ser grande, esta operación puede ser costosa. Sin embargo, no queda claro si puede ser costosa una y otra vez. En el ejemplo de la figura 6.26, la primera `obtenerMin` requiere 7 comparaciones, pero si se elimina ese elemento, sólo habrá tres candidatos para el siguiente mínimo. Todavía se desconoce la complejidad exacta de las operaciones con esta estructura de datos.

Veamos qué se requiere para implementar las demás operaciones de cola de prioridad. Los nodos pertenecerán a una clase organizadora, `nodoApareamiento`, que contiene por lo menos los campos `id` y `prioridad`. La inserción de un nodo nuevo es muy sencilla: creamos un árbol de un nodo y lo agregamos al bosque:

```
insertar(cp, v, w) // BOSQUEJO
   Crear nuevoNodo con id = v y prioridad = w.
   Arbol nuevoArbol = construirArbol(nuevoNodo, ListaArboles.nil);
   refx[v] = nuevoArbol;
   cp.bosque = cons(nuevoArbol, cp.bosque);
```

El borrado del mínimo, una vez hallado, simplemente convierte a todos sus subárboles principales en árboles del bosque.

```
borrarMin(cp) // BOSQUEJO
   obtenerMin(cp); // Asegurar que el bosque sólo tenga 1 árbol.
   Arbol t = primero(cp.bosque);
   cp.bosque = hijos(t);
```

El resultado podría ser un bosque vacío.

Para `decrementarClave`, necesitamos poder localizar el nodo, digamos `viejoNodo`, con base en su `id`. Un arreglo `refx`, como el que se usó para montones binarios en la sección 6.7.1, puede hacer eficiente esta operación. Es decir, `viejoNodo = raiz(refx[id])`. Ahora necesitamos evaluar el impacto de reducir la prioridad de un nodo dado. El nodo `viejoNodo` sigue siendo menor que todos los hijos de su propio subárbol; por tanto, si separamos todo el subárbol cuya raíz es `viejoNodo`, ese subárbol será un árbol en orden parcial válido por derecho propio, incluso después de reducirse la prioridad de su raíz. Podemos añadir este subárbol como árbol nuevo al Bosque de Apareamiento. Aunque el TDA `ListaArboles` no proporciona una operación para separar realmente un subárbol de un árbol de la lista, podemos establecer un valor especial en el campo `id` de `viejoNodo` para indicar que este árbol es obsoleto. Usamos -1 como

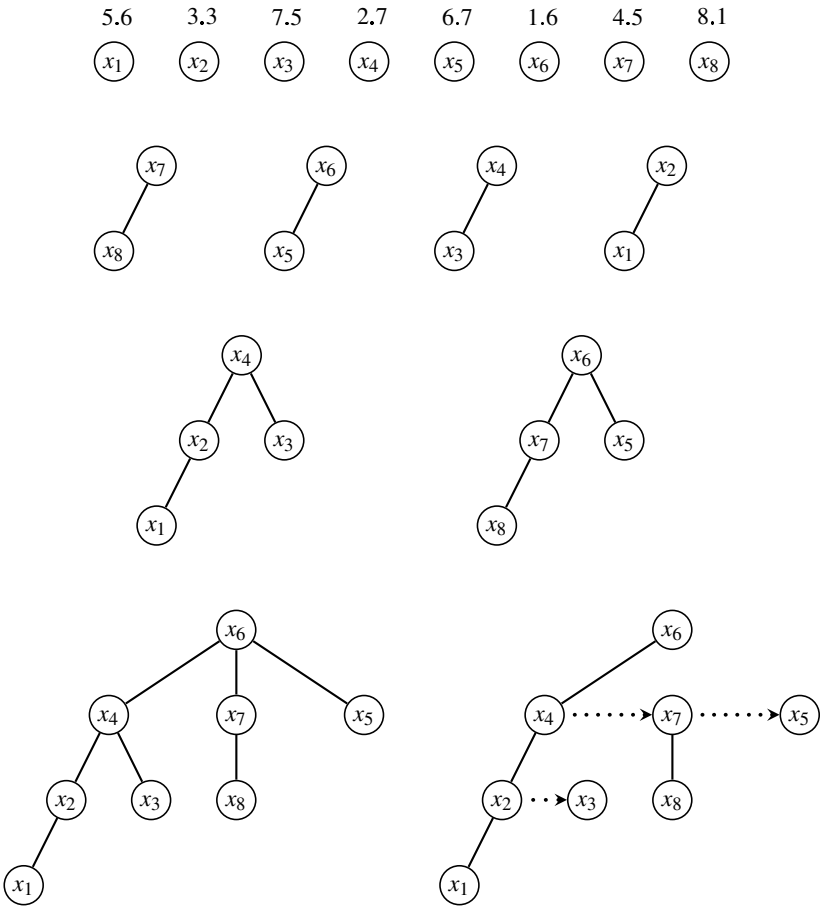


Figura 6.26 Ejemplo de torneo en el que se ejecutan operaciones `aparearArbol` con 8 raíces de árboles. En este ejemplo, los ocho árboles originales tienen un nodo cada uno, cuya prioridad se muestra arriba. Los pasos serían iguales si se tratara de las raíces de árboles más grandes. Cada perdedor se conecta al ganador como hijo de extrema izquierda y se invierte el orden de los ganadores en la lista. Por ejemplo, en la ronda 2, x_7 perdió ante x_6 y x_2 perdió ante x_4. Los ganadores de las primeras rondas se aparean en rondas posteriores. Después de tres rondas, se determina que la raíz mínima es x_6. La última fila muestra tanto la vista lógica, o conceptual, del árbol como la representación en la que los subárboles principales están en una lista, la cual presentamos en la sección 2.3.4. En la segunda representación, las aristas inclinadas hacia abajo van a los subárboles de extrema izquierda, mientras que las flechas horizontales van a subárboles hermanos derechos. En el último diagrama sólo los caminos que *comienzan* con una arista hacia abajo implican una relación de orden; así, x_4 debe ser menor o igual que x_3, pero no necesariamente tiene alguna relación con x_7 o x_8.

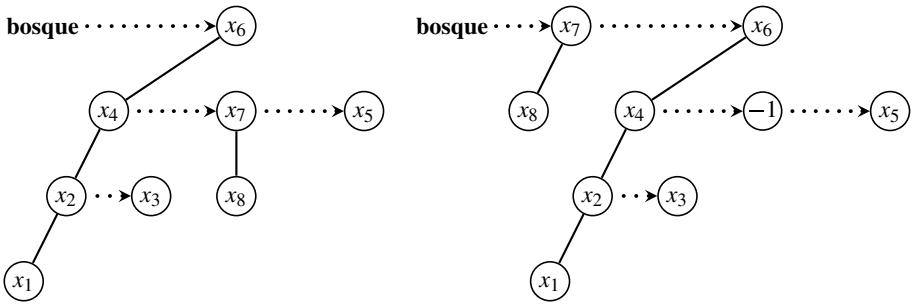


Figura 6.27 Ejemplo de `decrementarNodo` sobre el nodo x_7. Se usó la representación con los subárboles principales en una lista. El árbol que antes contenía a x_7 en su raíz todavía existe, pero ahora su raíz lo identifica como obsoleto. Antes de la operación, sabíamos que x_6 era menor o igual que x_7 en cuanto a valor de prioridad, pero ahora no conocemos ninguna relación entre ellos.

tal identificador especial. Supóngase que se ha creado `nuevoNodo` con los `id` y `prioridad` apropiados.

```
decrementarClave(cp, v, w) // BOSQUEJO
   Crear nuevoNodo con id = v y prioridad = w.
   Arbol viejoArbol = refx[v];
   NodoApareamiento viejoNodo = raiz(viejoArbol);
   Arbol nuevoArbol = construirArbol(nuevoNodo, hijos(viejoArbol));
   refx[v] = nuevoArbol;
   viejoNodo.id = −1; // Este árbol es obsoleto.
   cp.bosque = cons(nuevoArbol, cp.bosque);
```

La figura 6.27 muestra un ejemplo.

Siempre que se recorre una lista de árboles, simplemente se pasa por alto cualquier árbol cuya raíz tenga `id` $= -1$. Esto no implica una pérdida de eficiencia importante en el contexto de los bosques de apareamiento, porque la única lista de este tipo que se recorre es `bosque` misma, y la única ocasión en que se recorre se reconstruye como parte de la operación `obtenerMin`. (Además, `estaVacio(cp)` podría tener que pasar por, y desechar, nodos obsoletos hasta llegar a un nodo genuino.) Por tanto, los nodos obsoletos sólo se encuentran una vez, y en ese momento se desechan. Aunque falta precisar algunos detalles de implementación, son sencillos y se dejan para los ejercicios.

Análisis

Todas las operaciones se ejecutan en tiempo constante, con excepción de la combinación de `obtenerMin`, `borrarMin`. En casi todas las aplicaciones se invoca una inmediatamente después de la otra, por lo que es común integrarlas en una sola operación llamada `extraerMin`. Para los fines de este análisis, supondremos que así se hace. La operación `extractMin` aplicada a un bosque de k árboles se ejecuta en un tiempo proporcional a k, suponiendo que las operaciones del TDA de lista tardan un tiempo constante. Puesto que k puede ser n, el número de nodos de la co-

la de prioridad (y esto puede suceder con sólo insertar n nodos en una cola de prioridad vacía), el tiempo de peor caso para `extraerMin` está en $\Theta(n)$. Sin embargo, al igual que con las operaciones de Unión-Hallar que vimos en la sección 6.6.6, no puede darse el peor caso en todas y cada una de una serie de operaciones. Un análisis más minucioso requiere técnicas avanzadas que rebasan el alcance de este libro, así como algunas preguntas que todavía no tienen respuesta. Estudios empíricos indican que los Bosques de apareamiento son eficientes en la práctica. Remitimos a los lectores interesados a las Notas y referencias al final del capítulo.

Casi todas las aplicaciones que usarían un Bosque de apareamiento saben cuáles elementos estarán en la cola de prioridad, así que conocen n en el momento en que se crea el bosque de apareamiento. Las necesidades de espacio son proporcionales a n, sin importar cuántas operaciones se efectúen, siempre que los identificadores de los elementos no se salgan del intervalo $1, \ldots, n$.

Ejercicios

Sección 6.2 Doblado de arreglos

6.1 Evalúe el equilibrio de tiempo y espacio para la política de multiplicar el tamaño actual del arreglo por cuatro, en vez de por dos, cada vez que es preciso ampliar el arreglo. (Suponga que nunca se borran elementos.)

Sección 6.3 Análisis de tiempo amortizado

6.2 A fin de ahorrar espacio para una pila, se propone contraerla cuando su tamaño sea cierta fracción del número de celdas ocupadas. Esto complementa la estrategia de doblado de arreglos para ampliarla. Suponga que el costo es tn si hay n elementos en la pila, un costo similar al del doblado de arreglos.

Suponga que se mantiene la política de que el tamaño del arreglo se dobla cada vez que el tamaño de la pila rebasa el tamaño actual del arreglo. Evalúe cada una de las políticas de contracción propuestas siguientes, empleando costos amortizados si es posible. ¿Ofrecen un tiempo amortizado constante por operación? ¿Cuál esquema ofrece el factor constante más bajo? El tamaño actual del arreglo se denota con N.

a. Si un `pop` hace que haya menos de $N/2$ elementos en la pila, el arreglo se reduce a $N/2$ celdas.

b. Si un `pop` hace que haya menos de $N/4$ elementos en la pila, el arreglo se reduce a $N/4$ celdas.

c. Si un `pop` hace que haya menos de $N/4$ elementos en la pila, el arreglo se reduce a $N/2$ celdas.

*** d.** ¿Puede idear un esquema con parámetros distintos de los anteriores, que funcione aún mejor?

Sección 6.4 Árboles rojinegros

6.3 Demuestre que la tercera parte de la definición 6.1 es necesaria; es decir, dibuje un árbol que *no* tenga la propiedad de árbol de búsqueda binaria (definición 6.3) pero aun así satisfaga las partes 1 y 2 de la definición 6.1, y que al barrer una línea vertical de izquierda a derecha se vayan encontrando las claves en orden ascendente.

6.4 Dibuje todos los árboles RN_1 y RN_2 y todos los árboles CRN_1 y CRN_2.

6.5 Demuestre el lema 6.1.

6.6 Demuestre el lema 6.2.

6.7 ¿Por qué no sirve una inversión de color para reparar un cúmulo crítico de tres nodos?

6.8 Partiendo de un árbol rojinegro vacío, inserte una tras otra las claves 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

6.9 Encuentre una sucesión de 15 inserciones de nodos en un árbol rojinegro inicialmente vacío tal que el resultado final tenga una altura negra de 2.

6.10 Escriba estas subrutinas relacionadas con el color para el algoritmo 6.2:

a. La función `colorDe` que devuelve `negro` si su parámetro es un árbol vacío y devuelve el color de la raíz en los demás casos.

b. `invertirColor`, según la descripción de la sección 6.4.5.

6.11 Escriba las subrutinas `repararDer` y `reequilDer` para el algoritmo 6.2.

6.12 Demuestre el lema 6.4.

6.13 Exprese los cambios estructurales necesarios para reequilibrar después de una inserción (véase la figura 6.9) en términos de rotaciones. *Sugerencia:* La que finalmente será la raíz del subárbol modificado participa en cada rotación.

6.14 Borre nodos del árbol creado por el ejercicio 6.8 ajustándose a cada una de las reglas siguientes:

a. Borre lógicamente del árbol original cada nodo, con independencia de los demás.

b. De forma acumulativa, siempre borre lógicamente la raíz del árbol que queda después del borrado anterior.

c. De forma acumulativa, siempre borre lógicamente la clave más pequeña que quede en el árbol.

6.15 Borre nodos del árbol creado por el ejercicio 6.9, de forma acumulativa, siempre borrando lógicamente la clave más grande que quede en el árbol.

6.16 Exprese en términos de rotaciones los cambios estructurales necesarios para reequilibrar después de borrar (vea la figura 6.17).

6.17

a. ¿Insertar un nodo en un árbol rojinegro y luego borrarlo siempre produce el árbol original? Demuestre que así es, o presente un contraejemplo en el que no suceda así.

b. ¿Borrar un nodo hoja de un árbol rojinegro reinsertando después la misma clave siempre produce el árbol original? Demuestre que así es, o presente un contraejemplo en el que no suceda así.

Sección 6.5 Hashing (dispersión)

6.18 Detalle los procedimientos de direccionamiento abierto para buscar, insertar y borrar claves en una tabla de dispersión. ¿En qué difieren las condiciones en las que los ciclos terminan en cada uno de estos procedimientos? Tome en cuenta la posibilidad de que haya celdas "obsoletas".

6.19 El tipo de una tabla de dispersión H bajo direccionamiento cerrado es un arreglo de referencias a listas, y bajo direccionamiento abierto es un arreglo de claves. Suponga que una clave requiere una "palabra" de memoria y un nodo de lista ligada requiere dos palabras, una para la clave y una para una referencia a una lista. Considere cada uno de estos factores de carga con direccionamiento *cerrado*: 0.25, 0.5, 1.0, 2.0. Sea h_C el número de celdas de la tabla de dispersión con direccionamiento cerrado.

a. Estime el espacio total requerido, incluido espacio para listas, con direccionamiento cerrado. Luego, suponiendo que se usa *la misma cantidad* de espacio para una tabla de dispersión con direccionamiento abierto, determine los factores de carga correspondientes si se usa direccionamiento abierto.
b. Ahora suponga que una clave ocupa *cuatro* palabras y que un nodo de lista ocupa cinco palabras (cuatro para la clave y una para la referencia al resto de la lista), y repita la parte (a).

Sección 6.6 Relaciones de equivalencia dinámicas y programas Unión-Hallar

6.20 Escriba algoritmos para procesar una sucesión de instrucciones HACER y ES empleando una matriz para representar la relación de equivalencia. El conjunto subyacente tiene n elementos. Aproveche el hecho de que la relación es simétrica para evitar trabajo extra. ¿Cuántos elementos de la matriz se examinan o modifican en el peor caso al procesar una lista de m instrucciones?

6.21 Demuestre que un programa *Unión-Hallar* de longitud m ejecutado con un conjunto de n elementos no realiza más de $(n + m)n$ operaciones de liga si se implementa con la unión no ponderada y el hallar sencillo.

6.22 La unión ponderada, `unionP`, utiliza el número de nodos de un árbol como su peso. Sea `unionA` una implementación que usa la altura de un árbol como su peso y hace que el árbol de menor altura sea un subárbol del otro.

a. Escriba un algoritmo para `unionA`.
b. Demuestre que todos los programas *Unión-Hallar* construyen el mismo árbol usando `unionP` que el que construyen usando `unionA`, o bien presente un programa que produzca árboles distintos. (En ambas implementaciones, si los árboles tienen el mismo tamaño, conecte el primer árbol al segundo como subárbol.)
c. Determine la complejidad en el peor caso de los programas *Unión-Hallar* que usan la operación `hallar` sencilla (sin compresión de caminos) y `unionA`.

6.23 Presente un programa *Unión-Hallar* de tamaño n que requiera $\Theta(n \log n)$ tiempo empleando la operación `hallar` sencilla (sin compresión de caminos) y la unión ponderada (`unionP`).

6.24 Sea $S = \{1, 2, \ldots, 9\}$ y suponga que se usan `unionP` y `hallarCC`. (Si los árboles con raíz en t y u tienen el mismo tamaño, `union`(t, u) hace que u sea la raíz del nuevo árbol.) Dibuje los

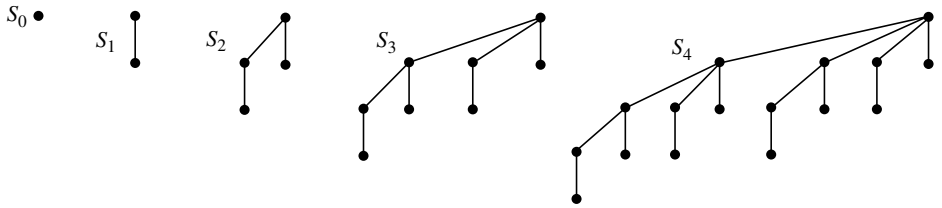


Figura 6.28 Árboles binomiales, también llamados árboles S_k

árboles después de la última `union` y después de cada `hallar` del programa siguiente. Para cada `hallar`, indique cuántos accesos `padre` (operaciones de liga) se usan.

```
union(1,2)
union(3,4)
union(2,4)
union(6,7)
union(8,9)
union(7,9)
union(4,9)
hallar(1)
hallar(4)
hallar(6)
hallar(2)
hallar(1)
```

6.25 Los árboles binomiales, también llamados árboles S_k, se definen como sigue: S_0 es un árbol de un nodo. Para $k > 0$, se obtiene un árbol S_k a partir de dos árboles S_{k-1} disjuntos conectando la raíz de uno a la raíz del otro. En la figura 6.28 se dan ejemplos.

Demuestre que, si T es un árbol S_k, T tiene 2^k vértices, altura k y un vértice único a la profundidad k. El nodo que está a la profundidad k es el *asa* del árbol S_k.

6.26 Utilizando las definiciones y los resultados del ejercicio 6.25, demuestre la siguiente caracterización de un árbol S_k: sea T un árbol S_k con asa v. Existen árboles disjuntos $T_0, T_1, \ldots, T_{k-1}$, que no contienen a v y tienen su raíz en $r_0, r_1, \ldots, r_{k-1}$, respectivamente, tales que

1. T_i es un árbol S_i, $0 \leq i \leq k - 1$, y
2. T es el resultado de conectar v a r_0, y r_i a r_{i+1}, para $0 \leq i < k - 1$.

Esta descomposición de un árbol S_4 se ilustra en la figura 6.29.

6.27 Utilizando las definiciones y resultados del ejercicio 6.25, demuestre la caracterización siguiente de un árbol S_k: sea T un árbol S_k con raíz r y asa v. Existen árboles disjuntos $T'_0, T'_1, \ldots, T'_{k-1}$ que no contienen r, con raíz en $r'_0, r'_1, \ldots, r'_{k-1}$, respectivamente, tales que

1. T'_i es un árbol S_i, $0 \leq i \leq k - 1$.

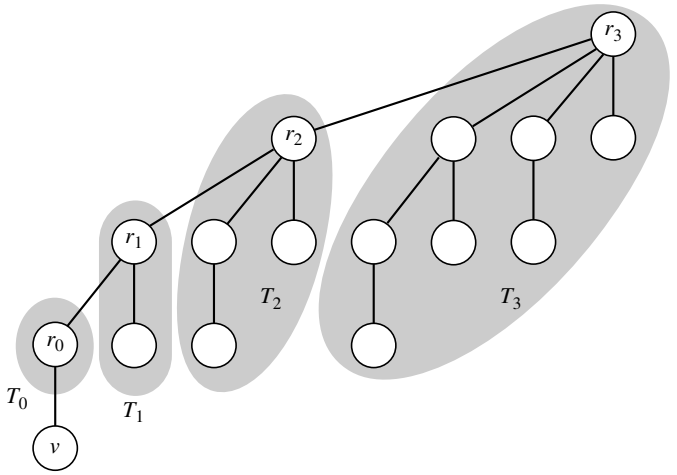


Figura 6.29 Descomposición de S_4 para el ejercicio 6.26

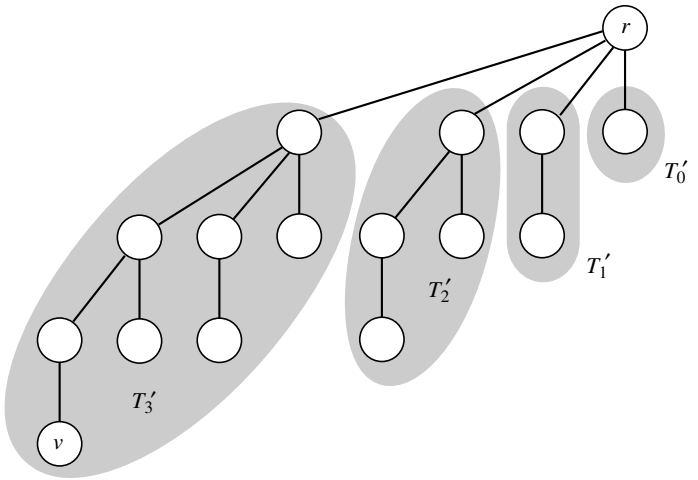


Figura 6.30 Descomposición de S_4 para el ejercicio 6.27

2. T se obtiene conectando cada r_i' a r, para $0 \leq i \leq k-1$, y
3. v es el asa de T_{k-1}'.

Esta descomposición de un árbol S_4 se ilustra en la figura 6.30.

★ **6.28** Una *incrustación* de un árbol T en un árbol U es una función uno a uno $f:T \rightarrow U$ (es decir, de los vértices de T a los vértices de U) tal que, para toda w y x en T, x es el padre de w si y sólo

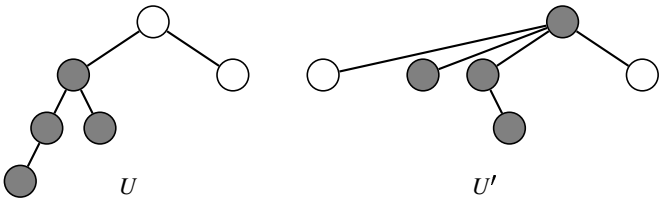


Figura 6.31 Incrustaciones de árboles binomiales para el ejercicio 6.28: los nodos sombreados están *incrustados propiamente* en U (izquierda) y están *incrustados inicialmente* en U' (derecha).

si $f(x)$ es el padre de $f(w)$. Una incrustación f es una *incrustación inicial* si establece una correspondencia entre la raíz de T y la raíz de U; de lo contrario, es una *incrustación propia*. En la figura 6.31 se dan ejemplos.

Utilizando los resultados de los ejercicios 6.25 a 6.27, demuestre que, si T es un árbol S_k con asa v, y f es una incrustación propia de T en un árbol U (que no tiene que ser binomial), y U' es el árbol resultado de ejecutar una `hallarCC` con $f(v)$ en U, entonces existe un árbol S_k T' incrustado inicialmente en U'. La figura 6.31 ilusttra el teorema; los nodos sombreados hacen las veces de T (izquierda) y T'(derecha).

⋆ **6.29** Demuestre que es posible construir un programa *Unión-Hallar* de longitud $m = n$ ejecutado con un conjunto de n elementos tal que, si se usan `hallarCC` y `unionNoP` para implementarlo, se efectuarán $\Omega(n \log n)$ operaciones. (*Sugerencia:* Lea los ejercicios 6.25 a 6.28.)

⋆ **6.30** Dijimos que hay ejemplos de programas *Unión-Hallar* que tardan más que un tiempo lineal incluso cuando se usan unión ponderada y hallar con compresión de caminos. Demuestre que, en un programa de longitud m ejecutado con un conjunto de n elementos, si todas las instrucciones `union` se ejecutan antes que todas las instrucciones `hallar`, el número total de operaciones estará en $O(n + m)$. *Sugerencia:* Modifique los costos contables de la definición 6.11.

⋆ **6.31** Suponga que relajamos el supuesto de que todos los conjuntos de un solo miembro se crean antes de que comience a ejecutarse el programa *Unión-Hallar*. Para tener la certeza de que nuestra "cuenta de ahorros" nunca se sobregire, necesitamos que la suma de los costos contables positivos sea por lo menos $4 \lg^*(n + 1)$ si el conjunto tiene actualmente n elementos. La k-ésima invocación de `hacerConjunto` puede presentarse en cualquier punto del programa. Puesto que no conocemos n, el número final de elementos, en el momento en que se invoca la k-ésima `hacerConjunto`, no podemos asignar a esta operación un costo contable basado en n. Sin embargo, sí conocemos k en el momento de la k-ésima `hacerConjunto`.

Demuestre que basta un costo contable de $4(2 + \lg^*(k + 1))$ para la k-ésima `hacerConjunto` para garantizar que la suma de costos contables positivos para n invocaciones de `hacerConjunto` sea por lo menos $4 \lg^*(n + 1)$.

6.32 Diseñe un algoritmo para procesar declaraciones **`equivalencia`** y asignar direcciones en memoria a todas las variables y arreglos de las declaraciones. Suponga que un enunciado **`dimension`** da las dimensiones de todos los arreglos. ¿Su algoritmo detecta declaraciones **`equivalencia`** no válidas?

Sección 6.7 Colas de prioridad con operación de decrementar clave

6.33 Muestre las etapas intermedias de la operación `decrementarClave` del ejemplo 6.13.

6.34 Muestre cómo evolucionan el montón y el arreglo `refx` durante una operación `borrarMin` aplicada al montón que se muestra a la izquierda en la figura 6.25.

6.35 Suponga que se insertan los elementos siguientes en un Bosque de apareamiento vacío en el orden dado: (1, 4.5), (2, 1.4), (3, 6.2), (4, 5.1), (5, 7.5), (6, 9.6), (7, 3.3), (8, 8.4), (9, 2.0). Cada elemento se escribe en la forma (`id`, `prioridad`). Cada una de las partes que siguen supone que se han efectuado las operaciones de las partes anteriores, de modo que los resultados son acumulativos. Tenga cuidado con el orden en todos los casos.

a. Muestre el Bosque de apareamiento después de las 9 inserciones anteriores.

b. Muestre el Bosque de apareamiento después de una `obtenerMin`, mostrando también los resultados intermedios después de cada `aparearBosque`.

c. Muestre el Bosque de apareamiento después de una `borrarMin`.

d. Muestre el Bosque de apareamiento después de que se ha reducido la prioridad de 7 a 2.2. No olvide incluir el nodo obsoleto.

e. Muestre el Bosque de apareamiento después de una segunda `obtenerMin`.

f. Muestre el Bosque de apareamiento después de una segunda `borrarMin`.

g. Muestre el Bosque de apareamiento después de una tercera `obtenerMin`.

6.36 Considere el algoritmo siguiente para hallar el segundo elemento más grande. Insertar todos los elementos en un Bosque de apareamiento *maximizante*. Ejecutar `obtenerMax`, luego `borrarMax`, luego `obtenerMax` otra vez. ¿Este algoritmo siempre efectúa un número óptimo de comparaciones cuando n, el número de elementos, es una potencia de 2?, es decir, ¿siempre iguala la cota inferior dada en el teorema 5.2? Demuestre que lo hace o dé un ejemplo de entrada con la que fracasa.

6.37 Recuerde que `decrementarClave` deja nodos obsoletos en el Bosque de apareamiento. Complete la implementación de estas operaciones de modo que detecten y desechen los nodos obsoletos.

a. `estaVacio(cp)`. ¿Su función se ejecuta en tiempo $O(1)$ en el peor caso? Si no, dé una sucesión de operaciones que obligue a `estaVacio` a requerir más que tiempo constante.

b. `aparearBosque(cp)`. ¿Su función se ejecuta en tiempo $O(k)$ en el peor caso, cuando en el bosque hay k árboles genuinos (es decir, árboles cuyas raíces no son obsoletas)?

c. Idee un esquema contable para la tarea de manejar nodos obsoletos. Suponga que desechar un árbol cuya raíz es obsoleta requiere una unidad de trabajo, que hacer obsoleta la raíz de un árbol requiere una unidad de trabajo, y verificar si la raíz de un árbol es obsoleta o no también requiere una unidad de trabajo. Haga caso omiso de todas las demás tareas, porque no tienen que ver con los nodos obsoletos. Asegure que los tiempos amortizados para `estaVacio` y `decrementarClave` estén en $O(1)$ según esta medida del trabajo, y que `aparearBosque` tenga un tiempo amortizado en $O(k)$ si hay k árboles genuinos en el bosque.

6.38 ¿Cuáles funciones y procedimientos para bosques de apareamiento necesitan actualizar el arreglo `refx`? Complete sus implementaciones. *Sugerencia:* Los elementos de `refx` son de tipo `Arbol`, así que hay que buscar lugares en los que se usa el constructor de `Arbol`.

6.39 La estrategia que describimos en el texto para `obtenerMin` es similar a la que se denomina *multipasadas* en otras obras. Una estrategia alterna, llamada *dospasadas*, funciona como sigue: se invoca `aparearBosque` una vez con el bosque inicial para dar un bosque intermedio, digamos $t_1, t_2, ..., t_j$. Cabe señalar que `aparearBosque` invierte el orden de la lista. Se invoca a `aparearArbol` con t_1 y t_2, para obtener el resultado w_2; luego se invoca `aparearArbol` con w_2 y t_3 para obtener el resultado w_3, y así sucesivamente. El resultado w_j tiene el elemento mínimo en su raíz. Se trata del método de "mínimo provisional" que acostumbra usarse para hallar el mínimo, con invocaciones de `aparearArbol` sobre la marcha. Precise los detalles de `obtenerMin2` utilizando esta estrategia. No olvide el arreglo `refx`.

Problemas adicionales

6.40 Evalúe la idoneidad de los árboles rojinegros para implementar tanto una cola de prioridad *elemental* como una cola de prioridad *completa*, tratando la clave de cada elemento como su prioridad. (Recuerde que una cola de prioridad elemental no incluye las operaciones `decrementarClave` ni `obtenerPrioridad`.) Considere el orden asintótico de peor caso de cada operación. En el caso de la cola de prioridad completa, se puede suponer que los elementos tienen identificadores dentro del intervalo 1, . . . , n. ¿Qué estructuras de datos auxiliares, como el arreglo `refx` mencionado en la sección 6.7, se necesitan para que `decrementarClave` sea eficiente? ¿Es directa su implementación? Si no, explique algunas complicaciones que se presenten.

Programas

1. Escriba un programa para implementar árboles rojinegros y probar sus operaciones. Incluya una opción para contar el número de comparaciones de claves, el número de inversiones de color y el número de reequilibraciones.
2. Escriba un programa para implementar el TDA Unión-Hallar empleando las operaciones `union` ponderada y `hallar` con compresión de caminos. El programa deberá probar las operaciones ejecutando algunos programas *Unión-Hallar*. Incluya una opción para contar el número de operaciones "de liga".
3. Escriba un programa para implementar un Bosque de apareamiento y pruebe sus operaciones. Incluya una opción para contar el número de comparaciones de claves. Cabe señalar que cada `aparearArbol` realiza una comparación de claves y las demás operaciones no realizan ninguna.

Notas y referencias

Los árboles rojinegros tienen una larga historia, habiéndose inventado con otros nombres y habiéndose redescubierto varias veces. La versión original se llamaba "árboles-B binarios simétricos" en Bayer (1972). El nombre *rojinegros* fue sugerido por Guibas y Sedgewick (1978), quie-

nes presentaron algoritmos de inserción y borrado descendentes que requerían $O(\log n)$ cambios estructurales (rotaciones). Otro nombre es "árboles 2-3-4". Tarjan (1983a, 1983b) escribió algoritmos para realizar reparaciones después de inserciones y eliminaciones con $O(1)$ cambios estructurales. Los métodos de reparación después de borrado presentados en este capítulo son un poco diferentes. Adoptamos el término *altura negra* de Cormen, Leiserson y Rivest (1990); Tarjan usó el término *rango*. Aho, Hopcroft y Ullman (1974) reseñan varios otros esquemas para mantener árboles binarios equilibrados, incluidos árboles AVL y árboles 2-3. Sleator y Tarjan (1985) introdujeron los árboles *splay* para el mismo fin. Los árboles *splay* son los más sencillos de implementar, pero los más difíciles de analizar, de todos los métodos de árbol equilibrado mencionados. No tienen un peor caso eficiente por operación, pero su costo amortizado es $O(\log n)$ por operación.

El hashing o dispersión se analiza a fondo en Knuth (1998). También se puede hallar un tratamiento exhaustivo en Cormen, Leiserson y Rivest (1990) y en Gonnet y Baeza-Yates (1991). Este último libro trata funciones de dispersión prácticas.

Van Leeuwen y Tarjan (1983) describen y analizan un gran número de técnicas para implementar programas *Unión-Hallar*, o de equivalencia. Galler y Fischer (1964) introdujeron el uso de estructuras de árbol para el problema de procesar declaraciones **`equivalencia`**. Knut (1968) describe el problema de equivalencia y presenta algunas sugerencias para encontrar una solución (véase su sección 2.3.3, ejercicio 11, también en Knuth (1997)). Fischer (1972) demuestra que, utilizando la unión no ponderada y el hallar con compresión de caminos, hay programas que efectúan $\Omega(n \log n)$ operaciones de liga. Los ejercicios 6.25 a 6.29 desarrollan la demostración de Fischer. M. Paterson (no publicado) demostró la cota superior de $O((m + n) \log n)$. Hopcroft y Ullman (1973) juntaron `unionP` y `hallarCC` y demostraron el teorema 6.13: es decir, que un programa de longitud m que se ejecuta con un conjunto de n elementos efectúa $O((m + n) \lg^*(n))$ operaciones. Tarjan (1975) estableció una cota inferior un poco mayor que lineal para el comportamiento de peor caso de `hallarCC` y `unionP` empleando como medida las operaciones de liga; Fredman y Saks (1989) generalizaron esto al modelo de cómputo de sonda de celda.

Se han efectuado investigaciones extensas acerca de estructuras de datos para colas de prioridad que hacen muy eficiente la operación `decrementarClave`. Fredman, Sedgewick, Sleator y Tarjan (1986) introdujeron los montones de apareamiento y describieron algunas variaciones. La versión que llamamos "Bosques de apareamiento" es similar a su método "multipasadas perezoso". El ejercicio 6.39 es similar a su "dospasadas". Ellos pudieron demostrar cotas un poco más categóricas para "dospasadas" que para "multipasadas". Jones (1986) informó estudios empíricos de varias estructuras de datos de cola de prioridad, y descubrió que los montones de apareamiento son competitivos. Stasko y Vitter (1987) realizaron estudios empíricos en los que "multipasadas" tuvo mejor desempeño que "dospasadas". Ellos introdujeron variaciones nuevas, llamadas "multipasadas auxiliar" y "dospasadas auxiliar". Dospasadas auxiliar tuvo el mejor desempeño de las cuatro variantes en sus experimentos, los autores también demostraron una cota de tiempo amortizado para esta variante más categórica que la que se ha demostrado para cualquiera de las otras variantes en la literatura, pero sólo para el caso en que no se usa `decrementarClave`.

Mientras tanto, Fredman y Tarjan (1987) introdujeron los montones de Fibonacci como estructura de datos para colas de prioridad y demostraron que tienen orden asintótico óptimo en términos de tiempo amortizado. Es decir, `decrementarClave` e `insertar` se ejecutan en $O(1)$, mientras que `borrarMin` se ejecuta en $O(\log n)$, en el sentido amortizado. Fredman (1999) aclaró en definitiva la duda que persistía desde hace mucho tiempo respecto a si los montones de apareamiento son o no óptimos en el sentido amortizado, demostrando que no tienen orden asintótico óptimo. El autor también presentó resultados empíricos adicionales y describió una clase

importante de aplicaciones en las que los montones de apareamiento "dospasadas" *sí tienen* orden asintótico óptimo. Sin embargo, todavía no se conocen los órdenes asintóticos exactos de todas las variaciones de los montones de apareamiento. A pesar de sus ventajas teóricas, los montones de Fibonacci se han descrito como un método cuya implementación es complicada y que implica un procesamiento fijo considerable, en comparación con los montones de apareamiento (Stasko y Vitter (1987), Fredman (1999)). Se pueden hallar reseñas de estructuras de datos para colas de prioridad en Cormen, Leiserson y Rivest (1990) y en Gonnet y Baeza-Yates (1991).

7

Grafos y recorridos de grafos

7.1 Introducción

Un grupo muy numeroso de problemas se puede plantear en algún tipo de grafo. Estos problemas surgen no sólo en computación, sino en todas las ciencias, la industria y los negocios. El desarrollo de algoritmos eficientes para resolver muchos problemas de grafos ha tenido un impacto considerable sobre nuestra capacidad para resolver problemas reales en todos esos campos. No obstante, para muchos problemas de grafos importantes todavía no se conocen soluciones. En otros casos, no se sabe si las soluciones que actualmente se conocen son lo más eficientes que podrían ser o si se les podría hacer mejoras sustanciales.

En este capítulo presentaremos las definiciones y propiedades básicas de los grafos. Luego nos ocuparemos de los métodos primordiales para recorrer grafos de manera eficiente. Resulta que muchos problemas naturales se pueden resolver con gran eficiencia —de hecho, en tiempo lineal— utilizando un recorrido de grafo como cimiento. En términos informales, podemos calificar a estos problemas de grafos como "fáciles"; no en el sentido de que fue fácil hallar o programar la solución, sino en el sentido de que, una vez programada, es posible resolver ejemplares del problema con gran eficiencia, y resulta práctico resolver ejemplares muy grandes del problema (grafos con millones de nodos, en algunos casos).

Para continuar con nuestra clasificación informal, la clase de problemas de grafos "de mediana dificultad" consiste en aquellos que se pueden resolver en tiempo polinómico, pero que requieren más trabajo que un recorrido elemental del grafo. Es decir, para cada problema "de mediana dificultad", se conoce un algoritmo que resuelve ejemplares de "tamaño" n en un tiempo acotado por arriba por algún polinomio fijo, como n^2, n^3 o n^d para alguna otra d fija. En capítulos posteriores trataremos varios problemas importantes de esta clase; véanse los capítulos 8, 9 y 14. En las potentes computadoras modernas resulta práctico resolver ejemplares relativamente grandes de estos problemas (grafos con millares o decenas de millares de nodos, digamos).

Más arriba en nuestra escala, tenemos los problemas de grafos "difíciles", para los cuales no se conoce ningún algoritmo de tiempo polinómico. En algunos casos, es imposible resolver problemas con grafos de 50 o 100 nodos con ningún algoritmo conocido, aunque se ejecute durante más de un año. No obstante, lo que sabemos hasta ahora no nos permite descartar la posibilidad de que se halle algún algoritmo eficiente. Estos problemas representan la verdadera frontera de nuestros conocimientos y veremos unos cuantos de ellos en el capítulo 13.

Uno de los aspectos fascinantes de los problemas de grafos es que cambios muy pequeños en la manera de plantear un problema a menudo pueden colocarlo en cualquiera de las tres categorías: fácil, de mediana dificultad o difícil. Por ello, familiarizarse con lo que se sabe acerca de los problemas existentes y de las características que los hacen fáciles, medianos o difíciles, puede ser muy útil al atacar un problema nuevo.

7.2 Definiciones y representaciones

En términos informales, un grafo es un conjunto finito de puntos (vértices o nodos), algunos de los cuales están conectados por líneas o flechas (aristas). Si las aristas no tienen dirección (son "bidireccionales"), decimos que el grafo es un *grafo no dirigido*. Si las aristas tienen una dirección (son "unidireccionales"), decimos que se trata de un *grafo dirigido*. Es común abreviar "grafo dirigido" a *digrafo*, a veces "grafo no dirigido" se abrevia a "grafo", pero esto puede ser ambiguo porque la gente a menudo se refiere a los grafos tanto dirigidos como no dirigidos simplemente como "grafos". Aquí usaremos el término específico en cualquier contexto en el que

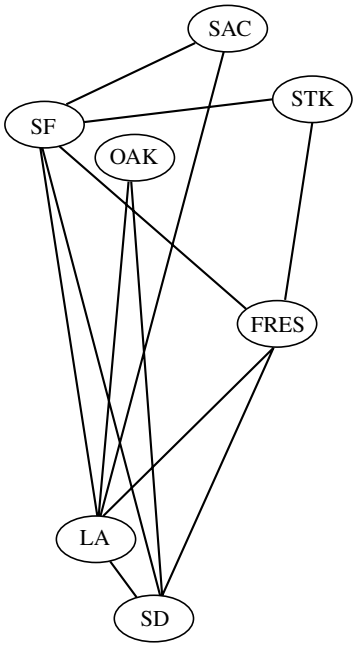


Figura 7.1 Grafo hipotético de vuelos sin escalas entre ciudades de California

pueda haber confusión. En explicaciones generales, "grafo" se refiere a los grafos tanto dirigidos como no dirigidos.

7.2.1 Algunos ejemplos

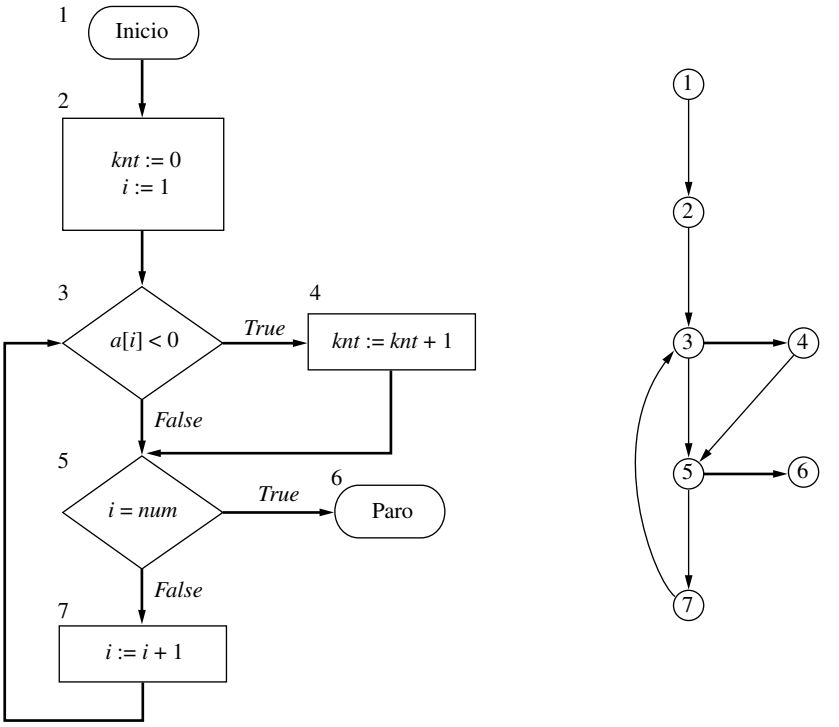
Los grafos son abstracciones útiles de numerosos problemas y estructuras en investigación de operaciones, ciencias de la computación, ingeniería eléctrica, economía, matemáticas, física, química, comunicaciones, teoría de juegos y muchas otras áreas. Consideremos los ejemplos siguientes:

Ejemplo 7.1 Mapa de rutas de aerolíneas

Un mapa de las rutas de una aerolínea se puede representar con un grafo no dirigido. Los puntos son las ciudades; una línea conecta dos ciudades si y sólo si hay un vuelo sin escalas entre ellas en ambas direcciones. En la figura 7.1 se muestra un mapa (hipotético) de rutas aéreas entre varias ciudades de California. ■

Ejemplo 7.2 Diagramas de flujo

Un diagrama de flujo representa el flujo de control en un procedimiento, o el flujo de datos o materiales en un proceso. Los puntos son los rectángulos del diagrama de flujo; las flechas que los conectan son las flechas del diagrama de flujo. En la figura 7.2 se muestra un ejemplo con sintaxis de Pascal. ■



(a) Diagrama de flujo (b) Grafo dirigido

Figura 7.2 Diagrama de flujo y el grafo dirigido correspondiente: las flechas indican la dirección del flujo

Ejemplo 7.3 Una relación binaria

Las relaciones binarias se definieron en la sección 1.3.1. Definimos R como la relación binaria sobre el conjunto $S = \{1, \ldots, 10\}$ que consiste en pares ordenados (x, y) en los que x es un factor propio de y; es decir, $x \neq y$ y el residuo de y/x es 0. Recordemos que xRy es una notación alterna para $(x, y) \in R$. En el grafo dirigido de la figura 7.3, los puntos son los elementos de S y hay una flecha de x a y si y sólo si xRy. Obsérvese que R es transitiva: si se cumplen tanto xRy como yRz, también se cumple xRz. ■

Ejemplo 7.4 Redes de computadoras

Los puntos son las computadoras. Las líneas (si el grafo es no dirigido) o las flechas (si el grafo es dirigido) son los enlaces de comunicación. La figura 7.4 muestra un ejemplo de cada caso. ■

Ejemplo 7.5 Un circuito eléctrico

Los puntos podrían ser diodos, transistores, condensadores, interruptores, etc. Dos puntos están conectados por una línea si hay una conexión eléctrica. ■

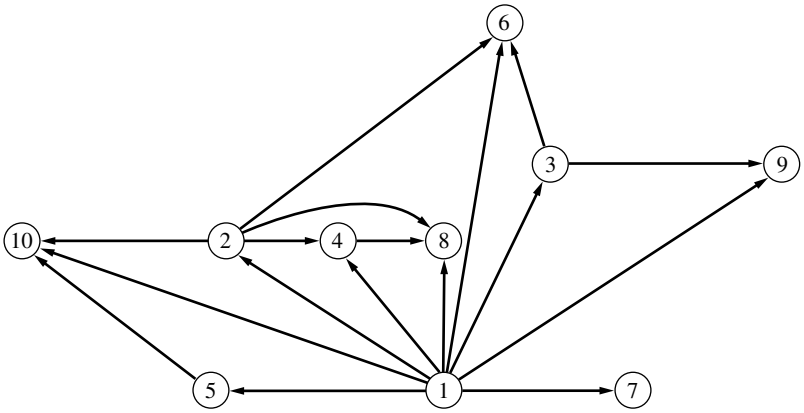
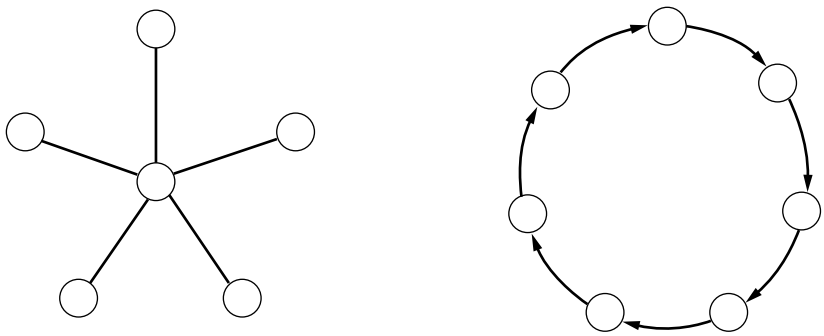


Figura 7.3 La relación *R* del ejemplo 7.3, que representa "*x* es un factor propio de *y*"



(a) Una red de estrella

(b) Una red de anillo

Figura 7.4 Redes de computadoras

Los cinco ejemplos anteriores deberán bastar para ilustrar el hecho de que los grafos dirigidos y no dirigidos son una abstracción natural de las relaciones entre diversos objetos, que incluyen tanto objetos físicos y su disposición, por ejemplo ciudades conectadas por rutas de aerolíneas, autopistas o líneas ferroviarias, como objetos abstractos, por ejemplo relaciones binarias y la estructura de control de un programa.

Estos ejemplos también deberán sugerir algunas de las preguntas que podría interesarnos hacer acerca de los objetos representados y de sus relaciones, preguntas que habrán de replantearse en términos del grafo. Tales preguntas pueden contestarse con algoritmos que trabajan con los grafos. Por ejemplo, la pregunta "¿Existe un vuelo sin escalas entre San Diego y Sacramento?" se traduce a "¿Existe una arista entre los vértices SD y SAC en la figura 7.1?" Consideremos las preguntas siguientes:

1. ¿Cuál es la forma más económica de volar de San Diego a Sacramento?
2. ¿Cuál ruta implica el menor tiempo de vuelo?

3. Si el aeropuerto de una ciudad está cerrado por mal tiempo, ¿seguirá siendo posible volar entre todos los demás pares de ciudades?
4. Si una computadora de una red se cae, ¿será posible enviar mensajes entre todos los demás pares de computadoras de la red?
5. ¿Qué tanto tráfico puede fluir de un punto dado a otro empleando ciertos caminos?
6. ¿Es transitiva cierta relación binaria?
7. ¿Un diagrama de flujo dado tiene ciclos?
8. ¿Cómo deben conectarse cables a diversos tomacorrientes de modo que la interconexión emplee la menor cantidad posible de cable?

En este capítulo y el que sigue estudiaremos algoritmos para contestar la mayor parte de estas preguntas.

7.2.2 Definiciones elementales de grafos

Esta sección está dedicada a definiciones y comentarios generales acerca de los grafos. Muchas afirmaciones y definiciones son válidas tanto para los grafos dirigidos como para los no dirigidos, usaremos una notación común para ambos a fin de reducir al mínimo la repetición. Sin embargo, ciertas definiciones son distintas para los grafos no dirigidos y los dirigidos, por tanto destacaremos tales diferencias.

Definición 7.1 Grafo dirigido

Un *grafo dirigido*, o *digrafo*, es un par $G = (V, E)$ donde V es un conjunto cuyos elementos se llaman *vértices* y E es un conjunto de pares *ordenados* de elementos de V. Los vértices también suelen llamarse *nodos*. Los elementos de E se llaman *aristas*, o *aristas dirigidas*, o *arcos*. Para la arista dirigida (v, w) en E, v es su *cola* y w es su *cabeza*; (v, w) se representa en los diagramas con la flecha $v \rightarrow w$. En el texto escribiremos simplemente vw. ■

En el ejemplo de relación binaria (ejemplo 7.3, figura 7.3),

$$V = \{1, 2, \ldots, 10\},$$

$$E = \{(1,2), \ldots, (1,10), (2,4), (2,6), (2,8), (2,10), (3,6), (3,9), (4,8), (5,10)\}.$$

Definición 7.2 Grafo no dirigido

Un *grafo no dirigido* es un par $G = (V, E)$ donde V es un conjunto cuyos elementos se llaman *vértices* y E es un conjunto de pares *no ordenados* de elementos distintos de V. Los vértices también suelen llamarse *nodos*. Los elementos de E se llaman *aristas*, o *aristas no dirigidas*, para hacer hincapié. Cada arista se puede considerar como un subconjunto de V que contiene dos elementos; así, $\{v, w\}$ denota una arista no dirigida. En los diagramas esta arista es la línea v-w. En el texto escribiremos simplemente vw. Desde luego, $vw = wv$ en el caso de grafos no dirigidos. ■

Para el grafo del ejemplo 7.1 y la figura 7.1, tenemos

$$V = \{\text{SF, OAK, SAC, STK, FRES, LA, SD}\},$$

$$E = \left\{ \begin{array}{llllll} \{\text{SF, STK}\}, & \{\text{SF, SAC}\}, & \{\text{SF, LA}\}, & \{\text{SF, SD}\}, & \{\text{SF, FRES}\}, & \{\text{SD, OAK}\}, \\ \{\text{SAC, LA}\}, & \{\text{LA, OAK}\}, & \{\text{LA, FRES}\}, & \{\text{LA, SD}\}, & \{\text{FRES, STK}\}, & \{\text{SD, FRES}\} \end{array} \right\}.$$

La definición de grafo no dirigido implica que no puede haber una arista que conecte un vértice consigo mismo: una arista se define como un conjunto que contiene dos elementos, y un conjunto no puede tener elementos repetidos, por definición (sección 1.3.1).

Definición 7.3 Subgrafo, grafo dirigido simétrico, grafo completo

Un *subgrafo* de un grafo $G = (V, E)$ es un grafo $G' = (V', E')$ tal que $V' \subseteq V$ y $E' \subseteq E$. Por la definición de "grafo", también es obligatorio que $E' \subseteq V \times V'$.

Un *grafo dirigido simétrico* es un grafo dirigido tal que, por cada arista vw existe también la arista inversa wv. Todo grafo no dirigido tiene un grafo dirigido simétrico correspondiente si se interpreta cada arista no dirigida como un par de aristas dirigidas en direcciones opuestas.

Un *grafo completo* es un grafo (normalmente no dirigido) que tiene una arista entre cada par de vértices.

Decimos que la arista vw *incide* en los vértices v y w, y viceversa. ■

Definición 7.4 Relación de adyacencia

Las aristas de un grafo dirigido o no dirigido $G = (V, E)$ inducen una relación llamada *relación de adyacencia*, A, sobre el conjunto de vértices. Sean v y w elementos de V. Entonces vAw (que se lee "w está *adyacente* a v") si y sólo si vw está en E. En otras palabras, vAw implica que es posible llegar a w desde v desplazándose a lo largo de una arista de G. Si G es un grafo no dirigido, la relación A es simétrica. (Es decir, wAv si y sólo si vAw.) ■

El concepto de camino es muy útil en muchas aplicaciones, incluidas algunas que implican la selección de rutas para personas, mensajes telefónicos (o electrónicos), tráfico de automóviles, líquidos o gases en tuberías, etc., y otras en las que los caminos representan propiedades abstractas (véase el ejercicio 7.3). Consideremos otra vez la figura 7.1 y supóngase que queremos volar de Los Angeles (LA) a Fresno (FRES). Existe una arista {LA, FRES} que podría ser una ruta, pero hay otras. Podríamos ir de LA a SAC a SF a FRES, o podríamos ir de LA a SD a FRES. Todos éstos son "caminos" de LA a FRES en el grafo.

Definición 7.5 Camino en un grafo

Un *camino de v a w* en un grafo $G = (V, E)$ es una sucesión de aristas $v_0v_1, v_1v_2, \ldots, v_{k-1}v_k$, tal que $v = v_0$ y $v_k = w$. La longitud del camino es k. Un vértice v sólo se considera un camino de longitud cero de v a v. Un *camino simple* es un camino tal que $v_0, v_1, \ldots, v_k$ son todos distintos.

Decimos que un vértice w es *asequible* desde v si existe un camino de v a w. ■

El camino {SD, FRES}, {FRES, SF}, {SF, SAC} se muestra en la figura 7.5. Denotamos un camino con una lista de la sucesión de vértices por la que pasa (pero recordando que la longitud de un camino es el número de aristas recorridas). Así, el camino de la figura 7.5 es SD, FRES, SF, SAC, y tiene una longitud de tres.

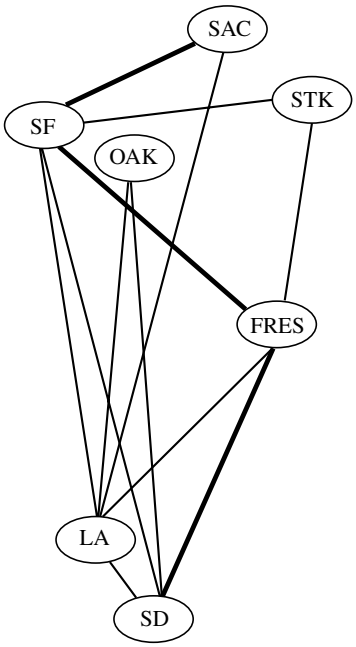


Figura 7.5 Camino de SD a SAC

Definición 7.6 Conectado, fuertemente conectado

Las definiciones de *conectividad* requieren atención porque difieren entre los grafos dirigidos y no dirigidos.

Un grafo no dirigido está *conectado* si y sólo si, para cada par de vértices v y w, existe un camino de v a w.

Un grafo dirigido está *fuertemente conectado* si y sólo si, para cada par de vértices v y w, existe un camino de v a w. ■

La razón por la que se dan definiciones distintas que al parecer son iguales es que, en un grafo no dirigido, si existe un camino de v a w, automáticamente existe un camino de w a v. En un grafo dirigido, esto podría no cumplirse, de ahí que se use el adverbio "fuertemente" para indicar que la condición es más categórica. Si vemos un grafo no dirigido como un sistema de calles de doble sentido y un grafo dirigido como un sistema de calles de un solo sentido, la condición de conectividad fuerte implica que podemos ir de cualquier lugar a cualquier lugar yendo por las calles de un solo sentido en la dirección correcta. Es evidente que esta condición es más estricta que si todas las calles fueran de doble sentido.

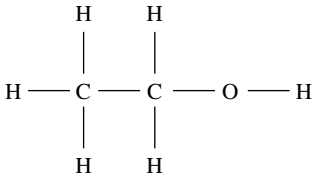


Figura 7.6 Árbol libre que representa una molécula de alcohol

Definición 7.7 Ciclo en un grafo

Las definiciones de *ciclos* requieren atención porque difieren entre los grafos dirigidos y no dirigidos.

En un grafo dirigido, un *ciclo* no es más que un camino no vacío tal que el primer vértice y el último sean el mismo, y un *ciclo simple* es un ciclo en el que ningún vértice se repite, con la salvedad de que el primero y el último son idénticos.

En los grafos no dirigidos las definiciones son similares, pero con el requisito adicional de que si cualquier arista aparece más de una vez, siempre aparece con la misma orientación. Es decir, empleando la notación de la definición 7.5, si $v_i = x$ y $v_{i+1} = y$ para $0 \leq i < k$, no puede haber una j tal que $v_j = y$ y $v_{j+1} = x$.

Un grafo es *acíclico* si no tiene ciclos.

Un grafo acíclico *no dirigido* se denomina *bosque no dirigido*. Si el grafo también está conectado, es un *árbol libre* o *árbol no dirigido*.

Es común usar la abreviatura *DAG* para referirse a un grafo acíclico dirigido (por sus siglas en inglés). (No se supone que un DAG satisfaga ninguna condición de conectividad.) ■

La figura 7.6 es un ejemplo de árbol libre, o no dirigido. Obsérvese que con esta definición de árbol no se señala a algún vértice específico como la raíz. Un *árbol con raíz* es un árbol en el que un vértice se ha designado como la raíz. Una vez especificada una raíz, es posible deducir las relaciones padre-hijo que suelen usarse con árboles.

La razón por la que distinguimos entre la definición de un *grafo dirigido simétrico* y la de un *grafo no dirigido* tiene que ver con los ciclos. Si el concepto de ciclos no es importante, por lo regular podremos usar un procedimiento diseñado para grafos dirigidos con el grafo dirigido simétrico que corresponde a un grafo no dirigido. En cambio, si los ciclos son importantes en el problema que nos ocupa, es poco probable que tal sustitución funcione. Por ejemplo, el grafo no dirigido simple que tiene la arista *ab* no tiene ciclos, pero su contraparte simétrica tiene dos aristas dirigidas, *ab* y *ba*, así que tiene un ciclo.

Definición 7.8 Componente conectado

Un *componente conectado* de un grafo no dirigido G es un subgrafo de G que es *máximo* y está conectado. En los grafos dirigidos el concepto correspondiente es más complejo, aplazaremos su definición hasta la definición 7.18. ■

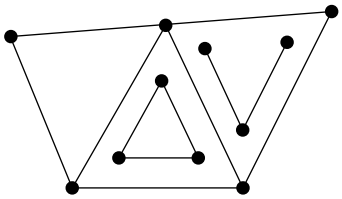


Figura 7.7 Grafo con tres componentes conectados

Debemos aclarar el significado de "máximo" en la definición de "componente conectado". Decimos que un grafo es *máximo* dentro de alguna colección de grafos si no es un subgrafo propio de ningún grafo de esa colección. No es necesario que sea el que más vértices tiene ni el que más aristas tiene entre los grafos de esa colección. En la definición 7.8 la "colección" es la de todos los subgrafos conectados de *G*.

Cuando se usa el término *componente* en relación con grafos y otras estructuras abstractas, por lo regular lleva la implicación de que es máximo dentro de algún grupo. Encontraremos los términos "componente fuertemente conectado" y "componente biconectado" más adelante en este capítulo. En ambos casos hay la noción de ser máximo.

Si un grafo no dirigido no está conectado, podría dividirse en componentes conectados individuales, siendo esta división única. El grafo de la figura 7.7 tiene tres componentes conectados.

En muchas aplicaciones de grafos es natural asociar un número, casi siempre llamado *peso*, a cada arista. Los números representan costos o beneficios que se derivan de dar algún uso a la arista en cuestión. Consideremos otra vez la figura 7.1 y supóngase que queremos volar de SD a SAC. No hay vuelos sin escalas, pero hay varias rutas o caminos que podrían usarse. ¿Cuál es mejor? Para contestar esta pregunta necesitamos una norma para juzgar los diversos caminos. La norma podría ser, por ejemplo,

1. el número de escalas;
2. el costo total del pasaje, y
3. el tiempo de vuelo total.

Después de escoger una norma, podríamos asignar a cada arista del grafo el costo (en escalas, dinero o tiempo) de viajar por esa arista. El costo total de un camino dado es la suma de los costos de las aristas que esa ruta recorre. En la figura 7.8 se muestra el grafo de aerolínea con el costo (hipotético) de un pasaje de avión escrito junto a cada arista. El lector puede verificar que la forma más económica de volar de SD a SAC es haciendo una escala en LA. El problema general de hallar caminos "óptimos" se estudia en las secciones 8.3 y 9.4.

La figura 7.9, que muestra algunas calles de una ciudad, podría servir para estudiar el flujo de tráfico automovilístico. El número asignado a una arista indica la cantidad de tráfico que puede fluir por esa sección de la calle en un intervalo de tiempo dado. La cifra depende del tipo y tamaño de la calle, el límite de velocidad, el número de semáforos entre las intersecciones que en el grafo aparecen como vértices (suponiendo que no se muestran todas las calles en el grafo), y varios otros factores.

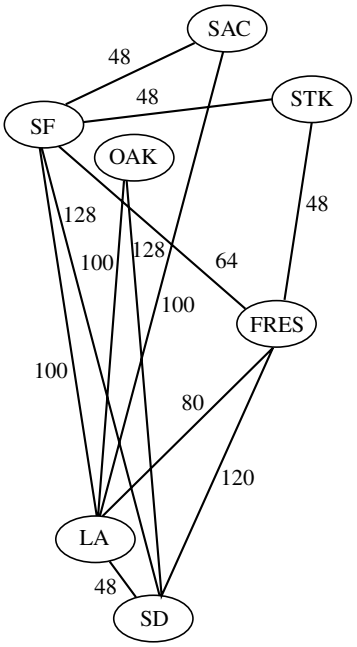


Figura 7.8 Grafo ponderado que muestra tarifas aéreas

La asignación de números a aristas se hace con suficiente frecuencia en las aplicaciones como para merecer una definición.

Definición 7.9 Grafo ponderado

Un *grafo ponderado* es una tripleta (V, E, W) en la que (V, E) es un grafo (dirigido o no dirigido) y W es una función de E sobre $\mathbf{R}$, los reales. (En algunos problemas podría ser apropiado que los pesos fueran de otro tipo, como números racionales o enteros.) Para una arista e, $W(e)$ es el *peso* de e. ■

La terminología funcional podría parecer muy técnica, pero es fácil entenderla una vez que recordamos lo visto en la sección 1.3.1, en lo conceptual, una función no es más que una tabla de dos columnas: el argumento de la función y el valor correspondiente de la función. En este caso, cada arista aparece en alguna fila en la columna 1 y su peso está en la misma fila en la columna 2. La representación en una estructura de datos podría ser distinta, pero comunicará la misma información. En diagramas de grafos, simplemente escribimos el peso junto a cada arista, como hicimos en las figuras 7.8 y 7.9. En algunas aplicaciones, los pesos corresponderán a costos o a aspectos indeseables de una arista, mientras que en otros los pesos serán capacidades u otras propiedades benéficas de las aristas. (La terminología varía con la aplicación; podrían usarse términos como *costo*, *longitud* o *capacidad* en lugar de *peso*.) En muchas aplicaciones, los pesos no pueden ser negativos por su naturaleza, como cuando representan distancias. La corrección de al-

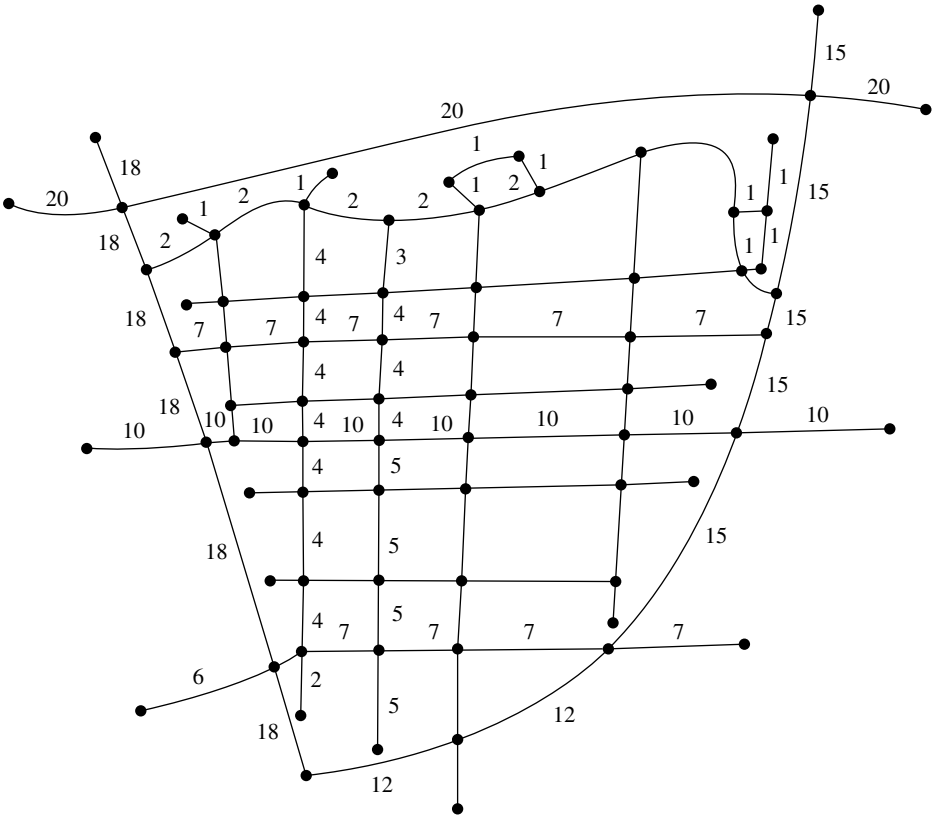


Figura 7.9 Mapa de calles que muestra capacidades de tráfico

gunos algoritmos depende de restringir los pesos a valores no negativos, mientras que otros algoritmos pueden manejar valores negativos.

7.2.3 Representaciones de grafos y estructuras de datos

Hemos visto dos formas de representar un grafo en papel: dibujando una imagen en la que los vértices se representan con puntos y las aristas con líneas o flechas, y haciendo una lista de los vértices y aristas. En esta sección veremos estructuras de datos que son útiles para representar grafos en un programa de computadora. Sea $G = (V, E)$ un grafo con $n = |V|$, $m = |E|$ y $V = \{v_1, v_2, \ldots, v_n\}$.

Representación de matriz de adyacencia

Podemos representar G con una matriz $A = (a_{ij})$ de $n \times n$ elementos, llamada *matriz de adyacencia* de G. A está definida por

$$a_{ij} = \begin{cases} 1 & \text{si } v_i v_j \in E \\ 0 & \text{en los demás casos} \end{cases} \qquad \text{para } 1 \leq i, j \leq n.$$

La matriz de adyacencia de un grafo no dirigido es simétrica (y sólo es preciso almacenar la mitad). Si $G = (V, E, W)$ es un grafo ponderado, los pesos se podrán almacenar en la matriz de adyacencia modificando su definición como sigue:

$$a_{ij} = \begin{cases} W(v_i v_j) & \text{si } v_i v_j \in E \\ c & \text{en los demás casos} \end{cases} \qquad \text{para } 1 \leq i, j \leq n.$$

donde c es una constante cuyo valor depende de la interpretación de los pesos y del problema a resolver. Si los pesos se ven como costos, se podría escoger ∞ (o algún número muy alto) para c porque el costo de recorrer una arista inexistente es prohibitivamente alto. Si los pesos son capacidades, suele ser apropiado escoger $c = 0$ porque nada puede viajar por una arista que no existe. Se dan ejemplos en las figuras 7.10(a, b) y 7.11(a, b).

Los algoritmos para resolver algunos problemas de grafos requieren examinar y procesar de alguna manera cada una de las aristas por lo menos una vez. Si se usa una representación de matriz de adyacencia, bien podríamos imaginar que un grafo tiene aristas entre todos los pares de vértices distintos, porque muchos algoritmos examinarían cada elemento de la matriz para determinar cuáles aristas existen realmente. Puesto que el número de posibles aristas es n^2 en un grafo dirigido, o de $n(n-1)/2$ en un grafo no dirigido, la complejidad de tales algoritmos estará en $\Omega(n^2)$.

Representación con arreglo de listas de adyacencia

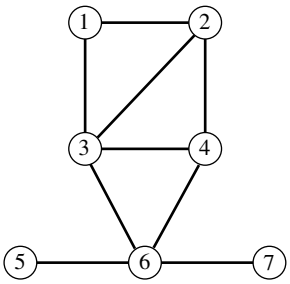
Una alternativa respecto a la representación con matriz de adyacencia es un arreglo indizado por número de vértice que contiene listas ligadas llamadas *listas de adyacencia*. Por cada vértice v_i, el i-ésimo elemento del arreglo contiene una lista con información acerca de todas las aristas de G que "salen de" v_i. En un grafo dirigido esto implica que v_i es la cola de la arista; en un grafo no dirigido, la arista incide en v_i. La lista de v_i contiene un elemento por arista. Para precisar la explicación, llamemos al arreglo `infoAdya`, que podría definirse así:

```
Lista[] infoAdya = new Lista[n+1];
```

Usaremos los índices $1, \ldots, n$, así que reservaremos espacio para $n + 1$ posiciones y no usaremos la posición 0. Ahora `infoAdya[i]` será una lista con información acerca de las aristas que salen de v_i.

La ventaja de una estructura de listas de adyacencia es que las aristas que no existen en G no existen tampoco en la representación. Si G es rala (es decir, si tiene muchas menos de n^2 aristas), se le podrá procesar rápidamente. Cabe señalar que si los elementos de una lista de adyacencia aparecen en un orden distinto, la estructura seguirá representando el mismo grafo, pero un algoritmo que use la lista encontrará los elementos en un orden distinto y podría comportarse de forma un tanto diferente. Los algoritmos no deben suponer algún orden específico (a menos, claro, que el algoritmo mismo construya la lista de alguna forma especial).

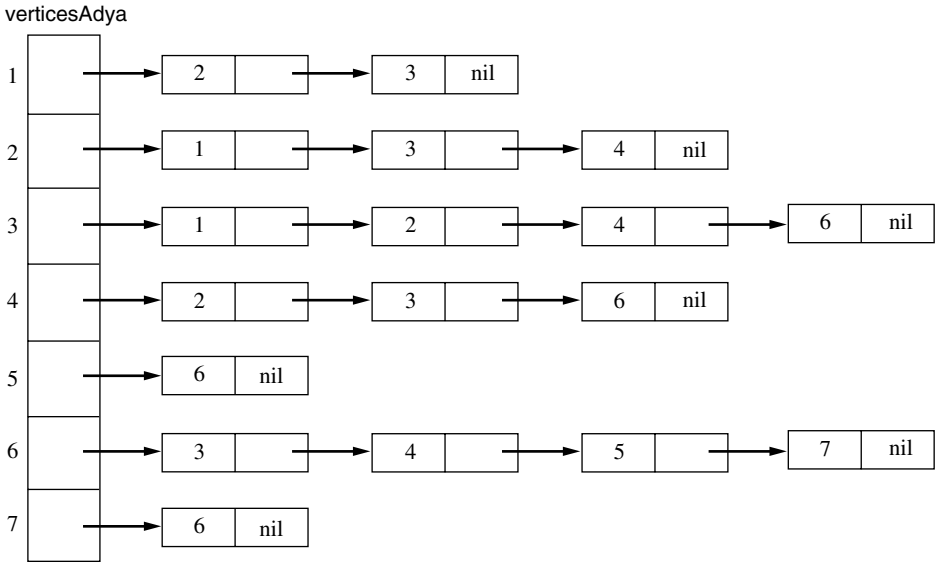
Los datos de las listas de adyacencia variarán según el problema, pero existen estructuras básicas más o menos estándar que son útiles para muchos algoritmos. Supóngase que definimos `InfoArista` como una clase organizadora (véase la sección 1.2.2) con campos para cada dato que queramos mantener acerca de la arista. Entonces, cada elemento de una lista de adyacencia será un objeto de la clase `InfoArista`. Tres campos comunes son `de`, `a` y `peso` para asentar que



(a) Un grafo no dirigido

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Su matriz de adyacencia

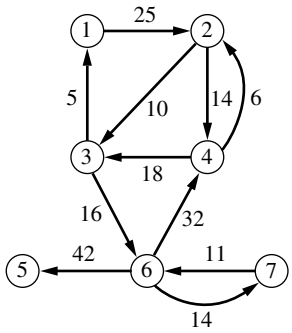


(c) Su estructura de listas de adyacencia

Figura 7.10 Dos representaciones de un grafo no dirigido sin pesos para las aristas son la matriz de adyacencia y el arreglo de listas de adyacencia. También podría ser un grafo dirigido simétrico.

la arista va de v_{de} a v_a y tiene un peso de "`peso`". Escribiremos esta información en la forma (`de`, `a`, `peso`). No obstante, en cualquier lista individual, el campo `de` será el mismo para todas las aristas. En particular, la lista `infoAdya[i]` tendrá `de = i` para todos sus elementos. Por tanto, el campo `de` es redundante y normalmente se omite de las listas de adyacencia.

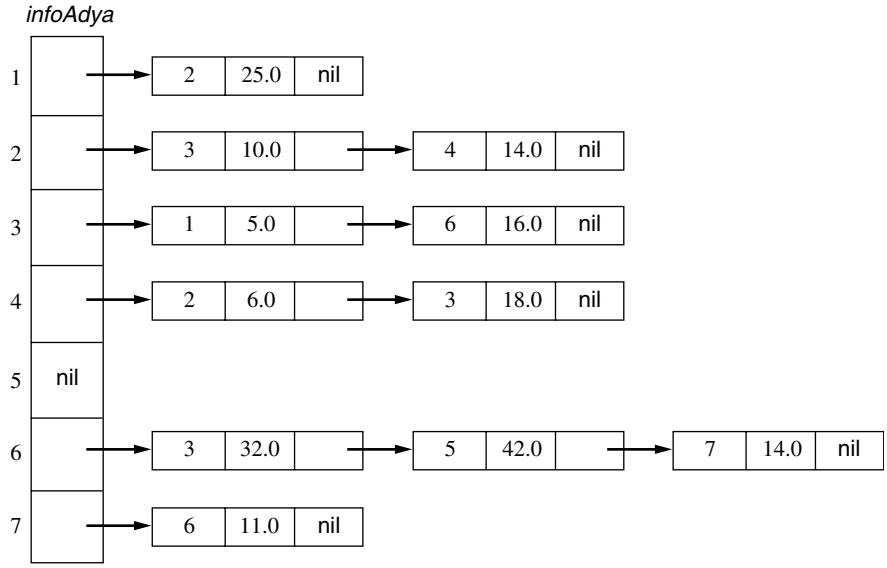
En el caso de grafos no dirigidos, tampoco habrá campo `peso`. Puesto que `InfoArista` se ha reducido a un solo campo, `a`, en este caso no necesitamos una clase organizadora. Simplemente usamos listas de enteros, como las que proporciona el tipo de datos abstracto `ListaInt` (sección 2.3.2). Puesto que ahora no hay más información que los vértices, cambiaremos el nombre del arreglo a `verticesAdya`. Cada elemento, digamos *j*, de la lista `verticesAdya[i]` indica la presencia de la arista $v_i v_j$ en *G*. Por ejemplo, si 6 está en la lista `verticesAdya[7]`, representa



(a) Un grafo dirigido ponderado

$$\begin{pmatrix} 0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\ 5.0 & \infty & 0 & \infty & 16.0 & \infty & \infty \\ \infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\ \infty & \infty & \infty & \infty & \infty & 11.0 & 0 \end{pmatrix}$$

(b) Su matriz de adyacencia



(c) Su estructura de listas de adyacencia

Figura 7.11 Dos representaciones de un grafo dirigido ponderado

la arista (7, 6). Esta estructura de datos se ilustra para un grafo no dirigido (que también podría ser un grafo dirigido simétrico) con el ejemplo de la figura 7.10.

En el caso de grafos ponderados, podríamos querer definir una clase de listas cuyos elementos están en `InfoArista` y llamar a esta clase `ListaAristas`. Denotemos a un objeto en `InfoArista` como (`a`, `peso`). En este caso, un elemento (j, w_{ij}) que está en la lista de adyacencia `infoAdya[i]` representa la arista (v_i, v_j) con peso w_{ij}. La figura 7.11 ilustra la estructura conceptual para un grafo dirigido ponderado. Podrían añadirse otros campos a los elementos del arreglo o a los elementos de las listas ligadas si así lo requieren los algoritmos que se usarán.

En un grafo no dirigido, cada arista se representa dos veces; es decir, si vw es una arista, hay un elemento para w en la lista de adyacencia para v, y un elemento para v en la lista de adyacen-

cia para w. Por tanto, cada lista de adyacencia tiene $2m$ elementos y hay n listas de adyacencia. En un grafo dirigido cada arista, al tener dirección, se representa una vez. Cabe señalar que las representaciones con listas de adyacencia de un grafo no dirigido y del *grafo dirigido simétrico* correspondiente son idénticas.

7.3 Recorrido de grafos

Casi todos los algoritmos para resolver problemas con un grafo examinan o procesan cada vértice y cada arista. La búsqueda primero en amplitud y la búsqueda primero en profundidad son dos estrategias de recorrido que permiten "visitar" de forma eficiente cada vértice y arista exactamente una vez. (Los términos *búsqueda primero en profundidad* y *recorrido primero en profundidad* son intercambiables; lo mismo los términos *búsqueda primero en amplitud* y *recorrido primero en amplitud*.) Por consiguiente, muchos algoritmos basados en tales estrategias se ejecutan en un tiempo que crece linealmente al crecer el tamaño del grafo de entrada.

7.3.1 Generalidades de la búsqueda primero en profundidad

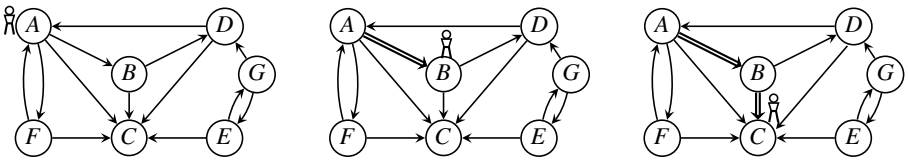
El valor de la búsqueda primero en profundidad fue puesto de manifiesto por John Hopcroft y Robert Tarjan, quienes desarrollaron muchos algoritmos importantes que lo usan. En el resto del capítulo presentaremos varios de esos algoritmos.

La búsqueda primero en profundidad es una generalización del recorrido general de un árbol (sección 2.3.4). El vértice inicial podría depender del problema o escogerse arbitrariamente. Al igual que con el recorrido de un árbol, resulta útil visualizar la búsqueda primero en profundidad como un viaje alrededor del grafo. La analogía con el recorrido de árboles se ve más claramente en el caso de los grafos dirigidos porque las aristas tienen una dirección, igual que las aristas de los árboles. Comenzaremos por describir la búsqueda primero en profundidad en grafos dirigidos y luego veremos cómo adaptarla a los grafos no dirigidos en la sección 7.6.

Imaginemos un grafo dirigido como un grupo de islas conectadas por puentes. Supondremos que el tráfico es en un solo sentido en cada puente, pero nuestro paseo será a pie, por lo que se nos permite caminar en ambas direcciones. No obstante, decidimos adoptar la política de que siempre cruzaremos un puente por *primera* vez en la dirección del tráfico; llamaremos a esto *explorar* una arista (puente). Si cruzamos un puente caminando en dirección opuesta al tráfico, estaremos regresando a algún lugar en el que ya estuvimos antes, por lo que llamaremos a esto *retroceder*. El tema de la búsqueda primero en profundidad es: explorar si es posible; si no, retroceder. Tenemos que añadir algunas restricciones sobre la exploración, pero lo haremos conforme "caminemos" por un ejemplo en la persona de Pepe el turista.

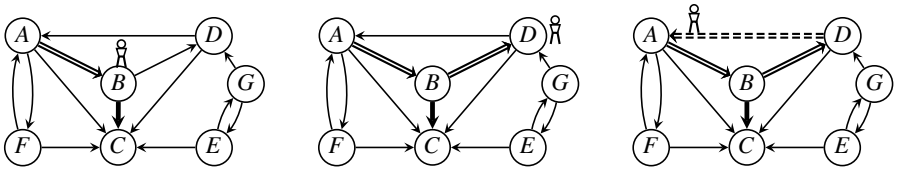
Ejemplo 7.6 Búsqueda primero en profundidad

Iniciemos una búsqueda primero en profundidad en el vértice A del grafo siguiente. Por sencillez, cuando podamos escoger qué arista explorar, las escogeremos en orden alfabético.

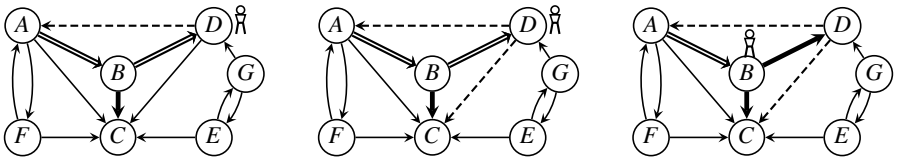


Pepe el turista parte de *A* en el diagrama de la izquierda, explora a *B* en el diagrama de enmedio y luego a *C* en el diagrama de la derecha. Las líneas continuas dobles denotan aristas que ya se exploraron y condujeron a vértices (o islas) aún no descubiertos. Decimos que *A*, *B* y *C* se *descubren* la primera vez que llega Pepe.

Recordando que la exploración debe efectuarse en la dirección del tráfico, vemos que desde *C* ya no hay a dónde explorar. Decimos que estamos en un *callejón sin salida*. Por tanto, Pepe *retrocede*. Los retrocesos siempre se efectúan usando el puente por el que se llegó a la isla la primera vez. Una vez que se retroceda desde la isla *C*, ésta no se volverá a visitar y decimos que está *terminada*. Una línea gruesa indica que una arista ya se exploró y se retrocedió por ella.



Arriba, en el diagrama de la izquierda, Pepe retrocedió a *B* y ahora aplica la regla de explorar si es posible. Todavía no ha explorado el puente a *D*, así que ése es el siguiente paso, que lleva al diagrama de enmedio. Ahora estamos en una situación que no puede presentarse al recorrer árboles. El diagrama de la derecha muestra a Pepe sobre el puente que lleva de *D* a *A*. Esto completaría un ciclo, pero por supuesto los árboles no tienen ciclos. Por esta razón, al buscar en un grafo, es necesario recordar dónde hemos estado: debemos poder distinguir entre los vértices no descubiertos y los descubiertos. Podríamos pasarnos la eternidad dando vueltas si no recordamos que ya descubrimos a *A*.



Supóngase que Pepe sí reconoce la isla *A* justo antes de llegar a ella y retrocede a *D*, como se muestra arriba en el diagrama de la izquierda. Usamos líneas punteadas para indicar que se exploró una arista, pero que conducía a un vértice ya descubierto.

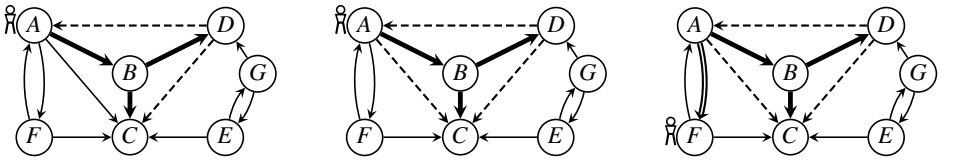
En la metáfora del viaje, decimos que semejante arista ya se exploró y se retrocedió por ella, aunque conduce a un vértice que ya se había descubierto. En cambio, cuando estemos pensando en la búsqueda algorítmicamente, diremos que semejante arista se *verificó*, y sólo usaremos el término *retroceder* cuando la arista se haya explorado y haya conducido a un vértice no descubierto.

De forma similar, exploramos el puente de *D* a *C*, pero *C* ya se descubrió e incluso se terminó, por lo que hay retroceso sin visita, lo que lleva al diagrama de enmedio. También decimos que *D* es un *callejón sin salida*, aunque salen aristas de él, porque sólo conducen a vértices descubiertos.

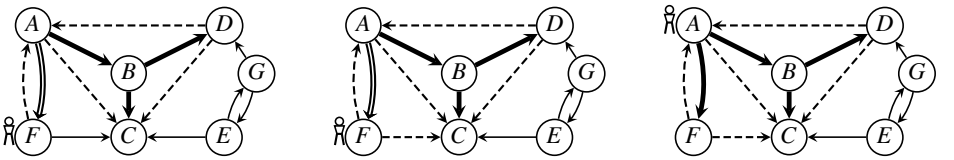
Observemos que, si bien ambas aristas *DA* y *DC* conducen a vértices descubiertos, hay una diferencia: la arista a *A* conduce a un vértice descubierto, pero no terminado, mientras que la aris-

ta a *C* conduce a un vértice terminado. Esta distinción es importante en muchas aplicaciones de la búsqueda primero en profundidad.

No hay más aristas que explorar desde *D*, así que Pepe retrocede por el puente que llevó al descubrimiento de *D* y vuelve a *B* (diagrama de la derecha). Ahí tampoco hay más puentes que explorar, así que el siguiente paso consiste en retroceder a *A*.



En el diagrama de la izquierda Pepe ha retrocedido a *A* y está listo para explorar en una nueva dirección. El puente *AC* es un tercer ejemplo de puente que conduce a un vértice ya descubierto, pero también hay una pequeña diferencia respecto a los otros dos. En este caso, exploraciones previas llevaron a Pepe de *A* a *C*, y ahora este puente es un atajo. En el caso de *DC* no había un camino que se hubiera recorrido antes de *D* a *C*. El diagrama de enmedio muestra la situación una vez que se ha explorado *AC* y se ha retrocedido por él. En el diagrama de la derecha Pepe ya exploró *AF* y llegó a un vértice no descubierto.



Desde la isla *F*, primero se explora *FA* y se retrocede por él (diagrama de la izquierda, arriba), luego se explora *FC* y se retrocede por él (diagrama de enmedio). Al igual que *D*, *F* es un callejón sin salida. Por último, Pepe retrocede a *A* para completar el paseo del día, como se muestra en el diagrama de la derecha. Observemos que Pepe nunca pudo llegar a *E* ni a *G*.

Si examinamos el último diagrama veremos que las aristas dibujadas con líneas gruesas continuas, que condujeron a vértices no descubiertos durante la búsqueda, forman un árbol. Esto es lógico si nos ponemos a pensar en ello, pues un vértice sólo se puede descubrir una vez, así que sólo puede llegar a él una arista de este tipo (o ninguna, si la búsqueda se inicia ahí). Que sólo una arista llegue a cada vértice es una propiedad de los árboles. El árbol definido por las aristas que llevaron a vértices no descubiertos durante la búsqueda se denomina *árbol de búsqueda primero en profundidad*, o árbol DFS (por sus siglas en inglés). Los árboles DFS se estudiarán más a fondo en la sección 7.4.3. ■

Aunque presentamos la búsqueda primero en profundidad como un viaje, nuestro ejemplo muestra que el viaje tiene cierta estructura: siempre regresamos por donde vinimos. En otras palabras, si el primer paso es de *A* a *B*, tarde o temprano regresaremos de *B* a *A*. ¿Qué sucedió mientras tanto? La realidad es que efectuamos una búsqueda primero en profundidad desde *B* con la condición adicional de que no podíamos volver a visitar a *A*.

En términos más generales, siempre que el viajero regresa (retrocede) a *A*, la condición adicional para continuar con la exploración es que no se vuelva a visitar *ningún* vértice que ya se

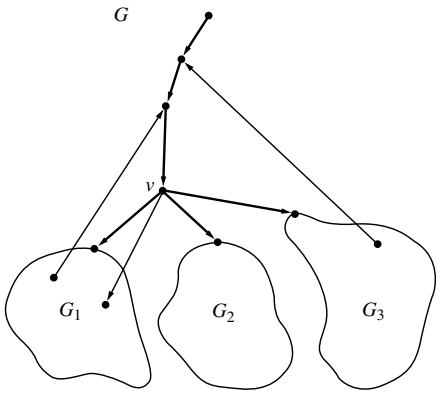


Figura 7.12 Estructura de la búsqueda primero en profundidad: se recorre G_1 totalmente antes de explorar G_2, luego G_3. Puesto que G podría no ser un árbol, podría haber aristas que van de los subgrafos a vértices que ya se visitaron antes.

haya visitado. Por ejemplo, la arista *AC* condujo a un vértice descubierto, pero la *AF* no, así que se efectuó una búsqueda primero en profundidad desde *F*. A causa de la regla que prohíbe volver a visitar un vértice que ya se descubrió, la exploración desde *F* no visitó *A* ni *C* antes de retroceder a *A*. En vez de ello, se *verificaron* las aristas *FA* y *FC*. Estas observaciones sugieren una descomposición recursiva del proceso de búsqueda:

```
dfs(G, v) // BOSQUEJO
    Marcar v como "descubierto".
    Para cada vértice w tal que la arista vw está en G:
        Si w no se ha descubierto:
            dfs(G, w); es decir, explorar vw, visitar w, explorar desde ahí hasta donde sea
            posible, y retroceder de w a v.
        Si no:
            "Verificar" vw sin visitar w.
    Marcar v como "terminado".
```

Para entender mejor la estructura de la búsqueda primero en profundidad, examinemos la figura 7.12. Supóngase que los vértices a los que se llegará desde v durante una búsqueda primero en profundidad se pueden dividir en varios subgrafos, G_1, G_2, G_3, tales que no existe conexión (a través de vértices no descubiertos) entre G_1, G_2 y G_3. También supondremos para este ejemplo que la lista de adyacencia de v está organizada de tal manera que algún vértice de G_1 se descubre antes que cualquier vértice de G_2, y algún vértice de G_2 se descubre antes que cualquier vértice de G_3.

La estrategia primero en profundidad de siempre explorar un camino lo más lejos posible antes de retroceder (y explorar caminos alternos lo más lejos posible antes de retroceder más) tiene el efecto de visitar todos los vértices de G_1 antes de pasar a un subgrafo nuevo adyacente a v, en este caso G_2 o G_3. Después se visitarán todos los vértices de G_2 antes de visitar cualquier vértice de G_3. Esto es análogo al recorrido de árboles, que visita todos los vértices de un subárbol antes de pasar al siguiente subárbol. Volveremos a esta analogía cuando estudiemos las propiedades de la búsqueda primero en profundidad con mayor detalle, en la sección 7.4.1.

Hasta aquí nos hemos concentrado en los grafos dirigidos. La búsqueda primero en profundidad se puede aplicar igualmente a los grafos no dirigidos, sólo que es preciso resolver antes cier-

ta ambigüedad acerca de la "dirección hacia adelante" y la "dirección hacia atrás" de las aristas, porque ahora éstas no están dirigidas. Volveremos a este problema en la sección 7.6.

Por último, necesitamos considerar el hecho de que no es forzoso que se pueda llegar a todos los vértices de un grafo desde el vértice en el que se inició una búsqueda primero en profundidad. Vimos esto con los vértices *E* y *G* en el ejemplo 7.6. El breve fragmento de pseudocódigo que sigue describe la forma de manejar esta situación.

```
barridoDfs(G) // BOSQUEJO
    Asignar a todos los vértices de G el valor inicial "no descubierto".
    Para cada vértice v ∈ G, en algún orden:
        Si v no se ha descubierto:
            dfs(G, v); es decir, realizar una búsqueda primero en profundidad que
            inicie (y termine) en v; cualesquier vértices descubiertos durante una
            visita de búsqueda primero en profundidad previa no se vuelven a visi-
            tar; todos los vértices visitados durante esta dfs se clasifican ahora co-
            mo "descubiertos".
```

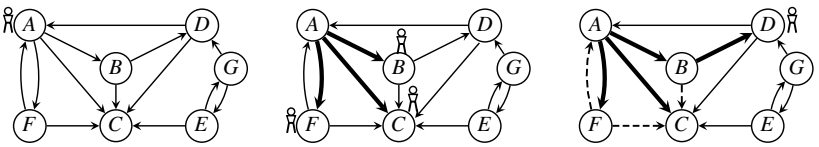
Dada la descripción informal de la búsqueda primero en profundidad, vemos que `barridoDfs` (mediante invocaciones de `dfs`) visita todos los vértices de *G* exactamente una vez, y recorre todas las aristas de *G* una vez en la dirección hacia adelante (explorando) y una vez en la dirección hacia atrás (retrocediendo). Sin embargo, cuando la arista conduce a un vértice que ya se descubrió, en lugar de decir que la arista se explora y de inmediato se retrocede por ella, decimos que la arista se *verifica*.

7.3.2 Generalidades de la búsqueda primero en amplitud

La búsqueda primero en amplitud es muy diferente de la búsqueda primero en profundidad en términos del orden en el que se descubren los vértices. En lugar de un viaje realizado por una persona, la mejor forma de visualizar la búsqueda primero en amplitud es como muchas exploraciones simultáneas (o casi simultáneas) que parten de un mismo punto y se extienden de manera independiente. Después de presentar una introducción informal, desarrollaremos un algoritmo de búsqueda primero en amplitud para una aplicación representativa, hallar un árbol abarcante primero en amplitud.

Ejemplo 7.7 Búsqueda primero en amplitud

Veamos cómo funciona la búsqueda primero en amplitud, partiendo del vértice *A* del mismo grafo que usamos en el ejemplo 7.6. En lugar de Pepe el turista, un *autobús lleno* de turistas deja a sus pasajeros en el punto *A*, desde donde comienzan a caminar en el diagrama de la izquierda. Los turistas se dispersan y exploran en todas las direcciones que permiten las aristas que salen de *A*, en busca de gangas. (Seguimos viendo a las aristas como puentes en un solo sentido, pero ahora también son en un solo sentido para los peatones, no sólo para el tráfico.)



Diversos grupos han llegado a *B*, *C* y *F* en el diagrama de enmedio. Supondremos que sólo el *primer* grupo que llega a una isla dada puede hallar las mejores gangas: compran y compran, y se las acaban. Aunque los turistas se siguen dispersando, sólo un contingente de *B* llega a un lugar no descubierto, *D*, como se muestra en el diagrama de la derecha.

Las líneas punteadas indican aristas que se exploraron pero conducían a vértices que ya se habían descubierto antes. (Una vez más, si estamos hablando de algoritmos, diremos que esas aristas se *verifican*, no se *exploran*.) Los turistas que siguieron esas rutas llegaron a *C* (o de regreso a *A*) demasiado tarde para conseguir gangas. Y no sólo eso; una vez que se atrasan de este modo, llegan demasiado tarde a cualesquier islas *futuras* a las que se podría llegar desde *A* o *C*, así que más les vale olvidarse de buscar gangas.

En la última fase de la búsqueda (que no se muestra), se exploran de forma similar las aristas *DA* y *DC*. En la búsqueda primero en amplitud no hay retrocesos y *E* y *G* son inalcanzables, así que la búsqueda terminará una vez exploradas estas dos últimas aristas.

Si examinamos el último diagrama, veremos que las aristas dibujadas con líneas gruesas continuas, que condujeron a vértices no descubiertos durante la búsqueda, forman otra vez un árbol, aunque es diferente del que se formó en el ejemplo 7.6. Si hay dos o más caminos más cortos a un vértice dado, el empate se romperá de alguna manera y sólo una arista se considerará como "descubridora" del vértice. La ganadora depende de los detalles de la implementación y de la estructura de datos al ejecutarse un programa de computadora. ■

Como vimos en el ejemplo, en la búsqueda primero en amplitud los vértices se visitan en orden de distancia creciente respecto al punto de partida, digamos *s*. La "distancia" en esta explicación es simplemente el número de aristas incluidas en un camino más corto. A continuación delinearemos el procedimiento con un poco más de detalle. En un principio ninguno de los vértices se ha descubierto.

El paso central de la búsqueda primero en amplitud, que parte de $d = 0$ y se repite hasta que dejan de hallarse vértices nuevos, consiste en considerar por turno cada vértice v que está a una distancia d de s y examinar todas las aristas que van desde v hacia vértices adyacentes. Para cada arista vw, si w no ha sido descubierto, se añade w al conjunto de vértices que están a una distancia $d + 1$ del punto de partida s; en caso contrario, w estará más cerca, y ya se conocerá su distancia.

Una vez procesados de esta manera todos los vértices que están a la distancia d, se procesan los vértices que están a la distancia $d + 1$ y así sucesivamente. La búsqueda termina cuando se llega a una distancia a la que ya no hay vértices.

Ejemplo 7.8 Distancias de primero en amplitud

Para la búsqueda primero en amplitud del ejemplo 7.7, las distancias son 0 para *A*, 1 para *B*, *C* y *F*, y 2 para *D*. En el ejercicio 7.5 se pide al lector calcular las distancias de primero en amplitud para el grafo del ejemplo 7.7, con *G* como vértice inicial. ■

Puesto que la búsqueda primero en amplitud tiene menos aplicaciones que la búsqueda primero en profundidad, concluiremos nuestra presentación de esa búsqueda aquí con una aplicación representativa. El algoritmo siguiente pone en práctica la búsqueda primero en amplitud que hemos descrito y halla un *árbol abarcante primero en amplitud* cuya raíz es un vértice inicial dado, *s*. El árbol se almacena como árbol adentro en el arreglo `padre`. Ya describimos el TDA Árbol Adentro en la sección 2.3.5, y vimos la implementación de un árbol adentro con arreglos en la sección 6.6.3.

Un *árbol abarcante primero en amplitud* contiene un vértice de árbol por cada vértice de grafo al que se puede llegar partiendo de s, de ahí el nombre "abarcante". Además, el camino en el árbol que va de s a cualquier vértice v contiene el número mínimo posible de aristas; por ello, la profundidad de v en este árbol es su distancia mínima en aristas respecto a s. En la parte final de la figura del ejemplo 7.7 las aristas gruesas continuas constituyen un árbol abarcante primero en amplitud.

Como siempre sucede en los árboles adentro, al camino que va del vértice inicial s a cualquier vértice v se puede descubrir en orden inverso siguiendo los elementos de `padre` de v hasta s. Se asigna el valor -1 al padre de s para indicar que s es la raíz.

Algoritmo 7.1 Búsqueda primero en amplitud

Entradas: $G = (V, E)$, un grafo representado por una estructura de listas de adyacencia, `verticesAdya`, como se describió en la sección 7.2.3, donde $V = \{1, \ldots, n\}$; $s \in V$, el vértice en el que se inicia la búsqueda.

Salidas: Un árbol abarcante primero en amplitud, almacenado en el arreglo `padre`. Ese arreglo se pasa como parámetro y el algoritmo se encarga de llenarlo.

Comentarios: Para una cola Q, suponemos que se usan operaciones del tipo de datos abstracto Cola (sección 2.4.2). El arreglo `color[1], ..., color[n]` denota la situación actual de todos los vértices respecto a la búsqueda. Los vértices no descubiertos son blancos; los que ya se descubrieron pero todavía no se procesan (en la cola) son grises; los que ya se procesaron son negros.

```
void busquedaPrimeroEnAmplitud(ListaInt[] verticesAdya, int n, int, s,
int[] padre)
    int[] color = new int[n+1];
    Cola pendiente = crear(n);
    Inicializar color[1], ..., color[n] con blanco.

    padre[s] = -1;
    color[s] = gris;
    encolar(pendiente, s);
    while (pendiente no esté vacío)
      v = frente(pendiente);
      desencolar(pendiente);
      Por cada vértice w de la lista verticesAdya[v];
        if (color[w] == blanco)
            color[w] = gris;
            encolar(pendiente, w);
            padre[w] = v; // Procesar la arista vw del árbol.
        // Seguir con la lista.
      // Procesar el vértice v aquí.
      color[v] = negro;
    return;
```

El algoritmo 7.1 sirve como esqueleto para todas las aplicaciones de búsqueda primero en amplitud. Los comentarios indican dónde se insertaría el código para procesar vértices y aristas de árbol (aristas a vértices no descubiertos, que constituyen el árbol abarcante primero en ampli-

tud). Si se desea procesar aristas no de árbol, se requerirá un `else` para el `if`; sin embargo, las búsquedas primero en amplitud raras veces tienen ese requisito.

Para la cola que se necesita en Búsqueda Primero en Amplitud (`pendiente`), la última parte del ejercicio 2.16 ofrece una implementación sencilla y eficiente, ya que sólo se efectuarán n operaciones de encolar durante el curso del algoritmo.

Como vimos en el ejemplo 7.7, no es forzoso que se pueda llegar a todos los vértices desde un vértice de partida dado. Si es necesario explorar todo el grafo, se puede usar un procedimiento de "barrido" similar a `barridoDfs` de la sección 7.3.1.

Análisis de Búsqueda Primero en Amplitud

Suponemos que G tiene n vértices y m aristas, y que la búsqueda llega a todo G. Además, suponemos que cada operación de cola tarda un tiempo constante. Por último, suponemos que el procesamiento que la aplicación efectúa con vértices y aristas individuales tarda un tiempo constante con cada uno; de lo contrario, sería necesario multiplicar los costos apropiados por el tiempo que tarda el procesamiento de cada operación.

Cada arista se procesa una vez en el ciclo **`while`** para dar un costo de $\Theta(m)$. Cada vértice se coloca en la cola, se saca de ella y se procesa una vez, para dar un costo de $\Theta(n)$. Se usa espacio extra para el arreglo `color` y la cola, y dicho espacio está en $\Theta(n)$.

7.3.3 Comparación de las búsquedas primero en profundidad y primero en amplitud

Antes de entrar en problemas y algoritmos específicos, demos un vistazo de alto nivel a algunas similitudes y diferencias de los dos métodos de recorrido que acabamos de bosquejar.

Las descripciones de los dos métodos de recorrido son un tanto ambiguas. Por ejemplo, si hay dos vértices adyacentes a v, ¿cuál se visitará primero? La respuesta depende de los detalles de la implementación; por ejemplo, de la forma en que se numeran o acomodan los vértices en la representación de G. Una implementación eficiente de cualquiera de los dos métodos debe mantenerse al tanto de cuáles vértices ya se descubrieron pero tienen vértices adyacentes que todavía no se han descubierto.

Cabe señalar que cuando una búsqueda primero en profundidad retrocede después de llegar a un callejón sin salida, supuestamente debe seguir otra rama desde el vértice descubierto *más recientemente* antes de explorar nuevos caminos que salen de vértices descubiertos hace más tiempo. Por tanto, los vértices desde los que la exploración es incompleta se procesan en orden de último en entrar primero en salir (LIFO), característico de una pila. Por otra parte, en una búsqueda primero en amplitud, a fin de garantizar que los vértices cercanos a v se visiten antes que los más lejano, los vértices a explorar se organizan en una cola FIFO.

Presentamos algoritmos de alto nivel para ambos métodos de búsqueda en las subsecciones anteriores. Es posible derivar muchas variaciones y extensiones de esos algoritmos, dependiendo del uso que se les dé. Por ejemplo, a menudo es necesario realizar algún tipo de procesamiento con cada arista. Las descripciones de los algoritmos no mencionan todas las aristas explícitamente, pero es obvio que la implementación de las líneas que requieren hallar un vértice no descubierto adyacente a un vértice dado, digamos v, implicaría examinar las aristas que inciden en v, y el procesamiento necesario de las aristas podría efectuarse ahí. En la sección 7.4.4 consideraremos cómo incorporar otros tipos de procesamiento en un esqueleto de búsqueda primero en profundidad general.

Para concluir esta comparación, observaremos que la búsqueda primero en profundidad contiene dos oportunidades de procesamiento para v (cuando se descubre y cuando se marca como "terminado"), mientras que la búsqueda primero en amplitud sólo contiene una (cuando se desencola). Si hacemos una inspección más minuciosa, notaremos que, en ambas búsquedas, la *primera* oportunidad de procesamiento se presenta mientras hay (posiblemente muchos) vértices no descubiertos a los que se puede llegar desde v. Por tanto, el tipo de cómputo que se puede efectuar en este punto deberá realizarse en un estado de relativa ignorancia acerca del resto del grafo. Por otra parte, en la búsqueda primero en profundidad hay también una oportunidad de procesamiento *en orden posterior*, justo antes de que la búsqueda retroceda por fin desde v. En ese momento, en general, se han descubierto muchos más vértices y se podría haber acumulado mucha más información durante la búsqueda. El paso de procesamiento en orden posterior a menudo puede aprovechar esta información adicional para realizar cálculos mucho más complejos que los que podían realizarse en la oportunidad de orden previo. La presencia de esta oportunidad de procesamiento en orden posterior en la búsqueda primero en profundidad explica, de forma muy profunda, por qué hay tantas aplicaciones de ese tipo de búsqueda y relativamente pocas de la búsqueda primero en amplitud.

7.4 Búsqueda de primero en profundidad en grafos dirigidos

Iniciaremos nuestro estudio detallado de la búsqueda primero en profundidad con los grafos dirigidos. Desarrollaremos un esqueleto general de búsqueda primero en profundidad que puede servir para resolver muchos problemas, y lo aplicaremos a varios problemas estándar.

El procedimiento general de búsqueda primero en profundidad es un poco más complicado para los grafos no dirigidos que para los dirigidos, por lo que nos ocuparemos de él en la sección 7.6. Esto podría parecer sorprendente porque los grafos no dirigidos al parecer son más sencillos que los grafos dirigidos. Sin embargo, una búsqueda primero en profundidad técnicamente correcta sólo explora cada arista una vez, y en los grafos no dirigidos cada arista está representada *dos veces* en la estructura de datos. Básicamente, una búsqueda primero en profundidad en un grafo no dirigido lo transforma en un grafo dirigido sobre la marcha, con cada arista orientada en la dirección de la exploración. Preferimos tratar esta cuestión aparte de los aspectos principales de la búsqueda primero en profundidad. También cabe señalar que varios problemas de grafos no dirigidos se pueden replantear como problemas de grafos dirigidos simétricos, en cuyo caso se podrá usar la búsqueda primero en profundidad dirigida, más sencilla. Como regla práctica, si la búsqueda primero en profundidad en los grafos no dirigidos hace caso omiso de las aristas, no de árbol, se puede usar el grafo dirigido simétrico correspondiente.

En muchos problemas en los que modelamos algo empleando un grafo dirigido, podría ser natural asignar aristas en cualquiera de las dos direcciones. Por ejemplo, consideremos un "grafo de invocaciones" en el que los vértices son procedimientos. Podría ser razonable definir las aristas empleando la regla de que vw significa "v invoca a w" o la regla de que vw significa "v es invocado por w". Como ejemplo adicional, consideremos un "diagrama genealógico" en el que los vértices son personas. Podría ser razonable definir las aristas en la dirección del padre al hijo o del hijo al padre. La opción más útil con toda seguridad dependerá del problema de que se trate. Por ello, es conveniente poder cambiar de una orientación a la otra. Ello justifica la definición siguiente.

Definición 7.10 Grafo transpuesto

El *grafo transpuesto* del grafo dirigido G, denotado por G^T, es el grafo que resulta de invertir la dirección de todas las aristas de G. ■

La búsqueda primero en profundidad explora en la dirección hacia adelante, pero hay algunos casos en que nos interesa buscar "hacia atrás" en un grafo. Es posible construir la estructura de listas de adyacencia del grafo transpuesto G^T a partir de la estructura de listas de adyacencia de G en tiempo lineal, para efectuar una búsqueda estándar de G^T. Como alternativa, si se prevé la necesidad en el momento de construir la estructura de listas de adyacencia, podrían construirse simultáneamente G y G^T.

7.4.1 Búsqueda primero en profundidad y recursión

Hemos visto que la búsqueda primero en profundidad se puede describir de manera sencilla con un algoritmo recursivo. De hecho, existe un vínculo fundamental entre la recursión y la búsqueda primero en profundidad. En un procedimiento recursivo, la estructura de invocaciones se puede diagramar como un árbol con raíz en el que cada vértice representa una invocación recursiva del procedimiento, como vimos en las secciones 3.2.1 y 3.7. El orden en el que se ejecutan las invocaciones corresponde a un recorrido primero en profundidad del árbol.

Ejemplo 7.9 Números de Fibonacci

Consideremos la definición recursiva de los números de Fibonacci, $F_n = F_{n-1} + F_{n-2}$, de la ecuación (1.13). En la figura 7.13 se muestra la estructura de invocaciones para un cálculo recursivo de F_6. Cada vértice se ha rotulado con el valor que tiene en ese momento n, el parámetro real del marco de activación que el vértice representa. En esencia, el subárbol cuya raíz es ese vértice calcula F_n para el valor actual de n.

En la figura se indica el orden de ejecución de las invocaciones recursivas; es la conocida sucesión en orden previo. Sin embargo, el orden en que la operación "+" acumula los resultados es

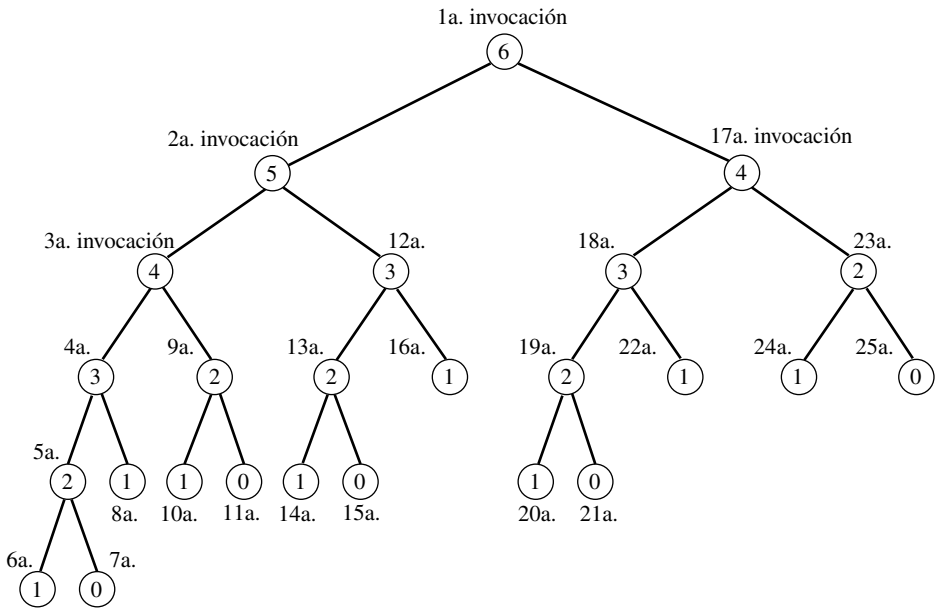


Figura 7.13 Estructura de invocaciones para el cálculo recursivo de números de Fibonacci

la sucesión en orden posterior. Ya vimos un ejemplo más pequeño con más detalles de los marcos de activación en el ejemplo 3.1.

Para el diagrama de la figura 7.13 supusimos que cada uno de los 25 vértices es distinto, aunque muchos tengan rótulos repetidos, porque cada vértice corresponde no sólo a su rótulo, sino también a una invocación de función específica. El lector tal vez sospeche que ésta es una forma en extremo ineficiente de calcular los números de Fibonacci, y tendría razón. Sería mucho más eficiente efectuar una búsqueda primero en profundidad en un grafo de siete vértices, cada uno con un rótulo único de 0 a 6. Volveremos a este tema en el capítulo 10, sección 10.2. Por ahora, sólo usaremos este ejemplo para ilustrar la relación entre la búsqueda primero en profundidad y la recursión. ■

Así pues, la estructura lógica de las soluciones de varios problemas interesantes que se resuelven con algoritmos recursivos es un recorrido primero en profundidad de un árbol. El árbol no siempre forma parte explícitamente del problema, y tampoco se representa explícitamente como estructura de datos. Como ejemplo adicional, examinemos el famoso problema de las ocho reinas.

Ejemplo 7.10 Ocho reinas en un tablero de ajedrez

Consideremos el problema de colocar ocho reinas en un tablero de ajedrez de forma que ninguna esté siendo atacada por ninguna otra; en otras palabras, de modo tal que ninguna puede llegar a otra desplazándose a lo largo de una fila, columna o diagonal. No es evidente que esto sea factible.

Lo intentaremos como sigue: colocamos una reina en el primer cuadrado (el de la extrema izquierda) de la primera fila (la de hasta arriba). Luego seguiremos colocando reinas en cada fila vacante sucesiva, en la primera columna que no esté amenazada por alguna reina ya colocada en el tablero. Continuaremos así hasta que las ocho reinas estén en el tablero o hasta llegar a una fila vacante que no tenga cuadrados no amenazados. Si se presenta este último caso (como sucede en la sexta fila; véase la figura 7.14), retrocedemos a la fila anterior, desplazamos la reina que está ahí el menor número posible de cuadrados a la derecha de modo que siga sin estar amenazada, y procederemos igual que antes.

¿Qué árbol implica este problema, y en qué sentido estamos efectuando una búsqueda primero en profundidad en él? El árbol se muestra en la figura 7.14. Cada vértice (distinto de la raíz) está rotulado con una posición del tablero. Para $1 \leq i \leq 8$, los vértices del nivel i se rotulan con posiciones de la fila i del tablero. Todos los hijos de un vértice v que están en el nivel i son posiciones del tablero en la fila $i + 1$ que no estarían amenazadas si hubiera reinas en todas las posiciones de tablero que están a lo largo del camino que va de la raíz a v; en otras palabras, los hijos son todos los cuadrados no amenazados de la fila siguiente. En términos del árbol, el problema consiste en hallar un camino de longitud 8 que vaya de la raíz a una hoja. Como ejercicio, el lector podría escribir un programa recursivo para el problema de las reinas tal que el orden en que se ejecutan las invocaciones recursivas corresponda a una búsqueda primero en profundidad. Si en verdad existe una solución, sólo se recorrerá parte del árbol de la figura 7.14. (Cuando la búsqueda primero en profundidad se usa en un problema de este tipo también se denomina búsqueda por retroceso.) ■

7.4.2 Identificación de componentes conectados con búsqueda primero en profundidad

En esta sección desarrollaremos de manera detallada un algoritmo para identificar los componentes conectados de un grafo, o los componentes fuertemente conectados de un grafo (dirigido) si-

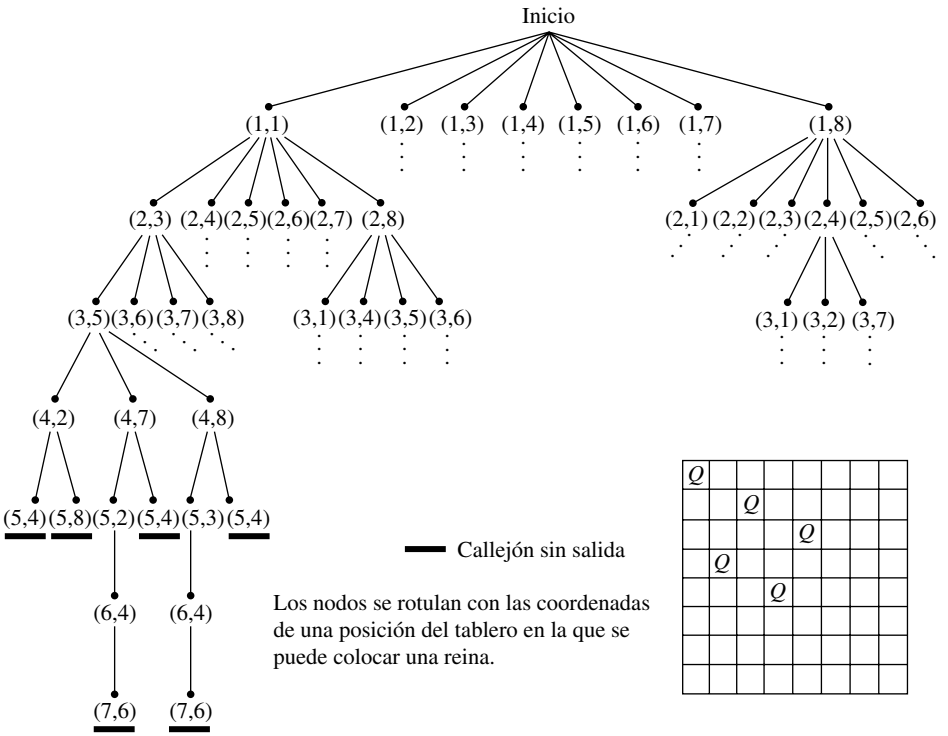


Figura 7.14 El problema de las ocho reinas

métrico. Al hacerlo, precisaremos varios detalles de implementación que se extienden a todas las aplicaciones de la búsqueda primero en profundidad. Los componentes conectados están asociados a los grafos no dirigidos, pero la representación de un grafo no dirigido es igual a la de un grafo dirigido simétrico, los componentes son los mismos en ambos casos. No obstante, el algoritmo de búsqueda primero en profundidad es un poco más sencillo en los grafos dirigidos.

Sea $G = (V, E)$ un grafo no dirigido con $n = |V|$ y $m = |E|$. El grafo dirigido simétrico correspondiente tiene $2m$ aristas dirigidas. Usaremos también G para ese grafo, pues sus representaciones son idénticas. Recordemos que un *componente conectado* de G es un subgrafo conectado máximo, es decir un subgrafo conectado que no está contenido en ningún subgrafo conectado más grande (definición 7.8). El grafo de la figura 7.7, por ejemplo, tiene tres componentes conectados. El problema de encontrar los componentes conectados de un grafo se puede resolver empleando búsqueda primero en profundidad casi sin adornos. Podemos partir de un vértice arbitrario, efectuar una búsqueda primero en profundidad para hallar todos los demás vértices (y aristas) del mismo componente y luego, si quedan vértices, escoger uno y repetir.

Usaremos del bosquejo de la búsqueda primero en profundidad (`dfs`) presentado en la sección 7.3.1. Varias partes del algoritmo podrían requerir mucho trabajo si escogemos una implementación poco apropiada. El ciclo necesita encontrar todos los w adyacentes a v (en la dirección hacia adelante de las aristas). Es indudable que conviene usar listas de adyacencia para represen-

tar el grafo, pues así podremos recorrer la lista de v y sólo examinar los w a los cuales llegan aristas desde v; si usáramos la matriz de adyacencia, tendríamos que examinar todos los w del grafo en este ciclo. En todo el algoritmo, sólo se recorre una vez cada lista de adyacencia. Usamos una variable local para recordar en qué punto de una lista de adyacencia estamos. (Esto implica que se guarda una variable en la pila de marcos para recordar en dónde nos quedamos en cada lista de adyacencia que hemos recorrido en parte, pero no totalmente, en cualquier momento dado, como se explicó en la sección 3.2.1.)

Como algoritmo, la búsqueda primero en profundidad opera en dos niveles. El nivel superior, o envoltura (`barridoDfs`), busca vértices no descubiertos e inicia una búsqueda primero en profundidad en cada vértice no descubierto que halla. El nivel inferior, llamado `dfs`, lleva a cabo recursivamente las acciones de una búsqueda primero en profundidad.

El problema de hallar un vértice no descubierto con `barridoDfs`, en el cual iniciar una nueva búsqueda primero en profundidad, se puede manejar de manera análoga a la forma en que `dfs` busca un nuevo vértice no descubierto. En lugar de examinar el arreglo de vértices desde el principio cada vez que llegamos al final de una búsqueda primero en profundidad, comenzamos en el punto donde nos quedamos la vez anterior.

Es indispensable tomar nota cada vez que la situación de un vértice cambia de "no descubierto" a "descubierto", a fin de evitar repetir trabajo y también para evitar la posibilidad de una búsqueda que no termine. En algunas aplicaciones también es importante tomar nota cuando se ha procesado cabalmente un vértice, es decir, cuando se ha "terminado". Esto también es muy útil para el análisis. Por ello, adoptaremos un sistema de tres colores para registrar la situación de los vértices.

Definición 7.11 Código de tres colores para la situación de búsqueda de los vértices

El color *blanco* denota que un vértice no se ha descubierto aún. El color *gris* denota que un vértice ya se descubrió pero su procesamiento todavía no termina. El color *negro* denota que un vértice ya se descubrió y terminó de procesarse. ■

Pasemos ahora a las necesidades específicas del problema de los componentes conectados. Si queremos dejar asentada en la estructura de datos, para uso futuro, la división del grafo en componentes conectados, podríamos hacerlo marcando cada vértice y/o arista con el número del componente al que pertenece. Una alternativa más compleja sería crear una lista ligada aparte con los vértices y/o aristas de cada componente. El método que se escoja dependerá de cómo se va a usar posteriormente la información.

A continuación presentamos el algoritmo de componentes conectados, el cual utiliza un procedimiento de búsqueda primero en profundidad que hace explícita la implementación. El procedimiento `componentesConectados` de este algoritmo corresponde al `barridoDfs` genérico que mencionamos antes y bosquejamos en la sección 7.3.1. Trataremos el grafo como grafo dirigido simétrico, no como grafo no dirigido, en el sentido de que efectuaremos una búsqueda primero en profundidad dirigida. La búsqueda primero en profundidad en un grafo no dirigido implica ciertas complicaciones adicionales que no son necesarias para hallar componentes conectados; nos ocuparemos de esos detalles en la sección 7.6.

Algoritmo 7.2 Componentes conectados

Entradas: Un arreglo `verticesAdya` de listas de adyacencia que representa un grafo dirigido simétrico $G = (V, E)$, según la descripción de la sección 7.2.3; y n, el número de vértices. El arreglo está definido para los índices $1, \ldots, n$. G también se puede interpretar como un grafo no dirigido.

Salidas: Un arreglo `cc` en el que cada vértice está numerado para indicar a qué componente pertenece. El identificador de cada componente conectado es el número de algún vértice incluido en ese componente (podrían usarse otros sistemas de identificación). (El invocador reserva espacio para `cc` y lo pasa como parámetro al procedimiento, que se encarga de llenarlo.)

Comentarios: Los significados de los colores son: blanco = no descubierto, gris = activo, negro = terminado. Cabe señalar que los parámetros tercero y cuarto de `ccDFS` son ambos v en la invocación de nivel más alto, pero tienen distinto significado. El tercer parámetro designa el vértice actual que se visitará cambiando en cada invocación recursiva. El cuarto parámetro designa el identificador del componente conectado y no cambia en las invocaciones recursivas.

```
void componentesConectados(ListaInt[] verticesAdya, int n, int[] cc)
    int[] color = new int[n+1];
    int v;
    Inicializar el arreglo color asignando blanco a todos los vértices.
    for (v = 1; v <= n; v ++)
      if (color[v] == blanco)
        ccDFS(verticesAdya, color, v, v, cc);
    return;

void ccDFS(ListaInt[] verticesAdya, int[] color, int v, int numCC,
int[] cc)
    int w;
    ListaInt adyaRest;

    color[v] = gris;
    cc[v] = numCC;
    adyaRest = verticesAdya[v];
    while (adyaRest ≠ nil)
      w = primero(adyaRest);
      if (color[w] == blanco)
        ccDFS(verticesAdya, color, w, numCC, cc);
      adyaRest = resto(adyaRest);
    color[v] = negro;
    return;
```

Análisis de componentes conectados

El número de operaciones efectuadas por `componentesConectados`, sin contar las invocaciones de `ccDFS`, es obviamente lineal en n. En `ccDFS`(..., v, ...), el número de instrucciones ejecutadas es proporcional al número de elementos de `verticesAdya[v]`, la lista de adyacencia recorrida, ya que la instrucción "`adyaRest = resto(adyaRest)`" se ejecuta una vez en cada iteración del ciclo **`while`**. Puesto que las listas de adyacencia sólo se recorren una vez, la complejidad de la búsqueda primero en profundidad, y por ende del algoritmo de componentes conectados, está en $\Theta(n + m)$. (Por lo regular, $m \geq n$.)

El espacio ocupado por la estructura de listas de adyacencia está en $\Theta(n + m)$, pero esto forma parte de las entradas del algoritmo. Se usa espacio adicional para el arreglo `color` ($n + 1$ ele-

mentos) y la recursión podría hacer que la pila de marcos de activación alcance un tamaño n, así que la cantidad de espacio extra utilizado está en $\Theta(n)$.

Comentarios acerca de componentes conectados

La salida del algoritmo es simplemente un arreglo `cc` que contiene un identificador (se usa con frecuencia el término *líder*) del componente conectado al que pertenece cada vértice. Basta una pasada por el arreglo `cc` para armar un conjunto de listas ligadas en el que cada lista contiene sólo los vértices de un componente conectado. Asimismo, basta una pasada por `verticesAdya` consultando `cc` para armar un conjunto de listas ligadas en el que cada lista contiene sólo las aristas de un componente conectado. Estos posibles pasos de postprocesamiento no aumentan la complejidad total, por lo que no tiene mucho caso complicar el algoritmo básico para adaptarlo a formatos de salida específicos.

El procedimiento `ccDFS` no efectuó ningún procesamiento de vértices en orden posterior (que consistiría en código colocado inmediatamente antes del enunciado "`color[v] = negro`"). Esto sugiere claramente que una búsqueda primero en amplitud también resolvería este problema con facilidad.

7.4.3 Árboles de búsqueda primero en profundidad

Los árboles de búsqueda primero en profundidad y el bosque de búsqueda primero en profundidad, que definiremos a continuación, ayudan a entender cosas importantes acerca de la estructura de la búsqueda primero en profundidad, que tiene muchos aspectos sutiles. El algoritmo 7.4 mostrará cómo construir árboles de búsqueda primero en profundidad. Pocos problemas requieren su construcción, pero de todos modos son útiles para los análisis. Las definiciones que damos aquí se refieren a grafos dirigidos. Aunque muchas de ellas son válidas en una forma similar a los grafos no dirigidos, a menudo hay diferencias en los pormenores, así que aplazaremos las definiciones para grafos no dirigidos hasta la sección 7.6.

Definición 7.12 Árbol de búsqueda primero en profundidad, bosque de búsqueda primero en profundidad

Las aristas que conducen a vértices no descubiertos (blancos) durante una búsqueda primero en profundidad en un grafo dirigido G forman un árbol con raíz llamado *árbol de búsqueda primero en profundidad* (también conocido como *árbol abarcante primero en profundidad*, y que se abrevia a *árbol DFS* en ambos casos). Si no es posible llegar a todos los vértices desde el vértice inicial (la raíz), un recorrido completo de G dividirá los vértices en varios árboles, que en conjunto forman el *árbol de búsqueda primero en profundidad* (también llamado *árbol abarcante primero en profundidad*, y que se abrevia a *bosque DFS* en ambos casos). ■

Definición 7.13 Decimos que un vértice v es *antepasado* de un vértice w en un árbol si v está en el camino que va desde la raíz hasta w; v es un antepasado *propio* de w si v es antepasado de w y $v \neq w$. El antepasado propio más cercano a v es el *padre* de v. Si v es un antepasado (propio) de w, entonces w es un descendiente (propio) de v. ■

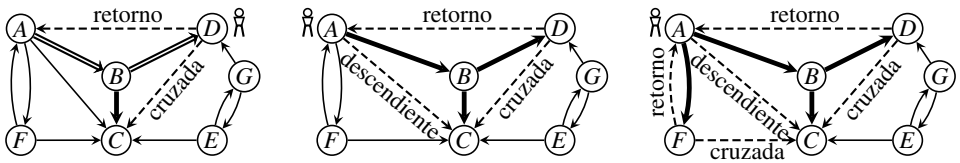
Definición 7.14

Las aristas de un grafo dirigido G se clasifican según la forma en que se *exploran* (se recorren en la dirección hacia adelante).

1. Si w no se ha descubierto aún en el momento en que se explora vw, decimos que vw es una *arista de árbol* y v se convierte en el padre de w.
2. Si w es antepasado de v, decimos que vw es una *arista de retorno*. (Esto incluye a vv.)
3. Si w es descendiente de v, pero w se descubrió antes de que se explorara vw, decimos que vw es una *arista descendiente* (también llamada *arista delantera* y *fronda*).
4. Si w no tiene una relación de antepasado/descendiente con v, decimos que vw es una *arista cruzada*. ■

Ejemplo 7.11 Árboles de búsqueda primero en profundidad

Veamos cómo se clasifican las aristas en la búsqueda primero en profundidad efectuada por Pepe el turista en el ejemplo 7.6. Pepe partió de A (así que A es la raíz del primer árbol de búsqueda primero en profundidad), exploró a B, luego a C, luego retrocedió a B y exploró a D. Entonces, las aristas AB, BC y BD son aristas de árbol. Ahora Pepe está en D y se topa con las primeras aristas no de árbol.



Las líneas dobles denotan aristas de árbol por las que todavía no se ha retrocedido, las líneas continuas gruesas denotan aristas de árbol por las que ya se retrocedió, las líneas delgadas son aristas inexploradas y las líneas punteadas son aristas no de árbol. En el diagrama de la izquierda A es un antepasado de D en el árbol, así que DA es una *arista de retorno*. En cambio, C no es antepasado ni descendiente de D en el árbol, así que DC es una *arista cruzada*.

El diagrama de enmedio muestra la situación después de que Pepe ha retrocedido de D a B y de B a A. El vértice C ya es un descendiente de A en el árbol en el momento en que se explora la arista AC; C se descubrió por otra ruta. Por tanto, AC es una arista *descendiente* (también llamada *arista delantera* o *fronda*). Una arista descendiente siempre es un atajo de un camino más largo en el árbol.

El diagrama de la derecha muestra la situación una vez que se ha completado el primer árbol de búsqueda primero en profundidad. Aunque Pepe no tiene a dónde ir, la búsqueda primero en profundidad del grafo está incompleta.

Para completar la búsqueda primero en profundidad del grafo, se inicia una nueva búsqueda en E. La arista EC conduce a un vértice que ya está terminado (negro) y que está en un árbol DFS que ya se completó. Es muy importante no volver a visitar C como parte del nuevo árbol DFS. La arista EC se clasifica como arista cruzada; obviamente, los vértices que están en árboles diferentes no tienen una relación de antepasado/descendiente. A continuación se considera la arista EG, la cual sí conduce a un vértice no descubierto (blanco), de modo que es una arista de árbol. La arista GD también conduce a un vértice terminado (negro) de un árbol DFS distinto, así que es una arista cruzada. La arista GE es una arista de retorno porque regresa a un antepasado de G en el árbol DFS actual. Así pues, el segundo árbol DFS, que completa el bosque DFS, tiene dos vértices y una arista. (También es posible tener un árbol con un vértice y ninguna arista: supóngase

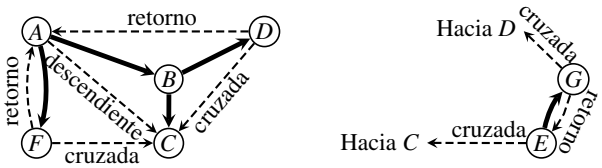


Figura 7.15 Las aristas gruesas muestran el bosque de búsqueda primero en profundidad para el grafo de los ejemplos 7.6 y 7.11. Las líneas punteadas son aristas no de árbol, y se clasifican como indican los rótulos.

que la primera búsqueda primero en profundidad del grafo se iniciara en el vértice *C*.) El bosque DFS final se muestra en la figura 7.15, junto con la clasificación de todas las aristas no de árbol. ■

Las clasificaciones de las aristas varían dependiendo del orden de los vértices dentro de una lista de adyacencia (véase el ejercicio 7.4). Cabe señalar que la cabeza y la cola de una arista cruzada podrían estar en árboles distintos. Las distinciones entre los diferentes tipos de aristas son importantes en algunas aplicaciones de búsqueda primero en profundidad; en particular, en los algoritmos que se estudian en las secciones 7.5 y 7.7.

7.4.4 Un esqueleto de búsqueda primero en profundidad generalizado

La búsqueda primero en profundidad proporciona una estructura para muchos algoritmos elegantes y eficientes. Como hemos visto en varios ejemplos, una búsqueda primero en profundidad encuentra cada vértice varias veces: cuando se descubre por primera vez el vértice y se convierte en parte del árbol de búsqueda primero en profundidad, luego varias veces más cuando la búsqueda retrocede *a* él y trata de seguir una dirección distinta, y al final, después del último de estos encuentros, cuando la búsqueda retrocede *del* vértice y no vuelve a pasar más por él. Dependiendo del problema a resolver, un algoritmo procesará los vértices de diferente manera al encontrarlos en las diversas etapas del recorrido. Muchos algoritmos efectúan también algún cálculo con las aristas; tal vez con cada una de las aristas, o tal vez sólo con las aristas del árbol de búsqueda primero en profundidad, o quizá diferentes tipos de cálculos con los diferentes tipos de aristas. El esqueleto de algoritmo que sigue muestra exactamente dónde se realizaría el procesamiento para cada tipo de arista y para cada tipo de encuentro con los vértices.

Algoritmo 7.3 Esqueleto de búsqueda primero en profundidad dirigida (esqueleto DFS)

Entradas: Un arreglo `verticesAdya` de listas de adyacencia que representa un grafo dirigido $G = (V, E)$, según la descripción de la sección 7.2.3; y n, el número de vértices. El arreglo está definido para los índices $1, \ldots, n$. Los demás parámetros serán los que necesite la aplicación.

Salidas: El valor devuelto depende de la aplicación. El tipo devuelto puede variar; **`int`** es sólo un ejemplo.

Comentarios: Este esqueleto también es apropiado para algunos problemas de grafos no dirigidos que hacen caso omiso de las aristas no de árbol, pero véase el algoritmo 7.8. Los significados de los colores son blanco = no descubierto, gris = activo, negro = terminado.

```
int barridoDfs(ListaInt[] verticesAdya, int n, ...)
    int respuesta;
    Reservar espacio para el arreglo color e inicializarlo con blanco.
    Para cada vértice v de G, en algún orden:
        if (color[v] == blanco)
          int vResp = dfs(verticesAdya, color, v, ...);
          (Procesar vResp)
        // Continuar el ciclo.
    return respuesta;

    int dfs(ListaInt[] verticesAdya, int[] color, int v, ...)
        int w;
        ListaInt adyaRest;
        int respuesta;
 1.     color[v] = gris;
 2.     Procesamiento en orden previo del vértice v
 3.     adyaRest = verticesAdya[v];
 4.     while (adyaRest ≠ nil)
 5.       w = primero(adyaRest);
 6.       if (color[w] == blanco)
 7.         Procesamiento exploratorio de la arista de árbol vw
 8.         int wResp = dfs(verticesAdya, color, w, ...);
 9.         Procesamiento al retroceder de la arista de árbol vw, usando wResp (como
            en orden interno)
10.       else
11.         Verificación (o sea, procesamiento) de la arista no de árbol vw
12.       adyaRest = resto(adyaRest)
13.    Procesamiento en orden posterior del vértice v, incluido el cálculo final de res-
       puesta
14.    color[v] = negro;
15.    return respuesta;
```

En algunas aplicaciones, el problema podría resolverse con una búsqueda parcial. Esta condición se detectaría en la línea 9 del esqueleto, o tal vez en la línea 11. Se recomienda un enunciado **break** para salir prematuramente del ciclo **while** pero efectuando el procesamiento en orden posterior, que incluye cambiar el color a negro.

Ejemplo 7.12 Uso del esqueleto DFS para componentes conectados

A fin de ilustrar la versatilidad del esqueleto, usémoslo para volver a resolver el problema de los componentes conectados.

1. Pasamos un arreglo `cc` como parámetro adicional de `barridoDfs`. El algoritmo llenará este arreglo con números de componentes conectados.
2. Añadimos un cuarto parámetro, `numCC`, y un quinto, `cc`, a `dfs`. En `barridoDfs`, al invocar `dfs`, asignamos el valor de v al cuarto parámetro, que también es el tercer parámetro, y asignamos el valor de `cc` al quinto parámetro.

3. En la invocación recursiva de `dfs`, usamos el mismo `numCC` y el mismo `cc` que se pasaron al procedimiento.
4. Como procesamiento en orden previo (línea 2 del esqueleto), insertamos el enunciado "`cc[v] = numCC`".

Son pocos cambios, pero con ellos, una línea de código nueva y el paso de unos cuantos parámetros más, hemos especializado el esqueleto de aplicación general para resolver el problema de los componentes conectados. Veremos más ejemplos del uso del esqueleto en el resto de este capítulo y en los ejercicios. ■

7.4.5 Estructura de la búsqueda primero en profundidad

En algunas aplicaciones de la búsqueda primero en profundidad, podría ser necesario saber cuáles vértices están en el camino que va desde la raíz del árbol DFS hasta el vértice actual, digamos v, que se está visitando. Ésos son exactamente los vértices grises y son los parámetros v de las invocaciones que están más abajo en la pila de marcos (es decir, más cerca de la raíz del árbol de marcos de activación). En algunos algoritmos se precisa conocer el orden en que se llega a los vértices por primera o por última vez, o relaciones entre los dos. Una forma sencilla y útil de seguir la pista a estas relaciones es manteniendo dos arreglos, `tiempoDescubrir` y `tiempoTerminar`. Se asigna el valor inicial cero a una variable global entera llamada `tiempo`, la cual se incrementará cada vez que cambie el color de un vértice.

Definición 7.15 Terminología de búsqueda primero en profundidad

Mientras el color de v es blanco, decimos que v está *ignoto*. Cuando `color[v]` cambia a gris, se asienta el valor actual de `tiempo` en `tiempoDescubrir[v]` (en la línea 2 del esqueleto); ahora v está *activo*. Cuando `color[v]` cambia a negro, se asienta el valor de `tiempo` en `tiempoTerminar[v]` (en la línea 13 del esqueleto), y ahora v está *terminado*. El *intervalo activo* del vértice v, denotado por *activo*(v), se define como el intervalo entero

$$activo(v) = \texttt{tiempoDescubrir[v]}, \ldots, \texttt{tiempoTerminar[v]}$$

que incluye ambos extremos, de modo que v es gris precisamente durante su intervalo activo. El valor final de `tiempo` será $2n$ si la búsqueda abarca todo el grafo. ■

Podemos insertar en el esqueleto DFS código para calcular tiempos de descubrimiento y término, para construir el bosque de búsqueda primero en profundidad. Llamamos a este algoritmo *Rastreo de Búsqueda Primero en Profundidad*. Aunque un algoritmo que use el esqueleto DFS no incluya este código, para fines de análisis podemos usar los valores que se habrían calculado si se hubiera insertado el código.

Algoritmo 7.4 Rastreo de búsqueda primero en profundidad (rastreo DFS)

Entradas: Las mismas que para el esqueleto DFS (algoritmo 7.3) más los arreglos globales `tiempoDescubrir`, `tiempoTerminar` y `padre`; también un contador global, `tiempo`.

Salidas: El algoritmo llena los arreglos globales antes mencionados. Los tipos devueltos por este algoritmo se pueden cambiar a **`void`**. En el arreglo `padre` se almacena el bosque de búsqueda primero en profundidad en forma de árbol adentro. Los demás arreglos tienen los significados descritos en la definición 7.15.

Estrategia: Modificamos el esqueleto DFS del algoritmo 7.3 como sigue:

1. En `barridoDfs` inicializamos `tiempo` con 0.

2. En `barridoDfs`, antes de invocar `dfs` (después del "if"), insertamos "`padre[v] = -1`".
3. En `dfs`, antes de la invocación recursiva de `dfs` (después del "if"), insertamos "`padre[w] = v`".
4. En el procesamiento en orden previo (línea 2) del esqueleto insertamos

```
tiempo ++; tiempoDescubrir[v] = tiempo;
```

(Éste es el tiempo en que v se vuelve activo.)

5. En el procesamiento en orden posterior (línea 13) del esqueleto insertamos

```
tiempo ++; tiempoTerminar[v] = tiempo;
```

(Éste es el tiempo en que v se vuelve inactivo.) ■

La figura 7.16 muestra un ejemplo del algoritmo 7.4. Los intervalos durante los cuales los vértices están *activos*, según la definición 7.15, se muestran como pares d/f, donde d es el tiempo de descubrimiento y f es el tiempo de finalización del vértice. Estos intervalos tienen una interesante e importante relación mutua y con las posiciones relativas de los vértices en el bosque de búsqueda primero en profundidad (definición 7.12).

Ejemplo 7.13 Anidación de intervalos *activos*

Para el bosque de búsqueda primero en profundidad de la figura 7.16, los intervalos *activos* se muestran en la figura 7.17. El vértice A es la raíz de un árbol de búsqueda primero en profundidad, y E es la raíz del otro. Sus intervalos *activo* son disjuntos. Obsérvese que todas las aristas cruzadas van de intervalos tardíos a intervalos anteriores no traslapantes. Además, donde hay una arista descendiente AC, hay un vértice B cuyo intervalo contiene el intervalo de C y está contenido en el intervalo de A. ■

Resumamos las relaciones ilustradas con el ejemplo anterior.

Teorema 7.1 Defínase *activo*(v) como en la definición 7.15, defínanse las clasificaciones de aristas como en la definición 7.14, y supóngase que se ha efectuado un rastreo DFS con un grafo $G = (V, E)$. Entonces, para cualquier $v \in V$ y $w \in V$,

1. w es un descendiente de v en el bosque DFS si y sólo si *activo*(w) $\subseteq$ *activo*(v). Si $w \neq v$, la inclusión es propia.
2. Si v y w no tienen una relación antepasado/descendiente en el bosque DFS, sus intervalos *activo* son disjuntos.
3. Si la arista $vw \in E$, entonces:
   a. vw es una arista cruzada si y sólo si *activo*(w) precede totalmente a *activo*(v).
   b. vw es una arista descendiente si y sólo si hay algún tercer vértice x tal que *activo*(w) $\subset$ *activo*(x) $\subset$ *activo*(v).
   c. vw es una arista de árbol si y sólo si *activo*(w) $\subset$ *activo*(v), y *no* existe un tercer vértice x tal que *activo*(w) $\subset$ *activo*(x) $\subset$ *activo*(v).
   d. vw es una arista de retorno si y sólo si *activo*(v) $\subset$ *activo*(w).

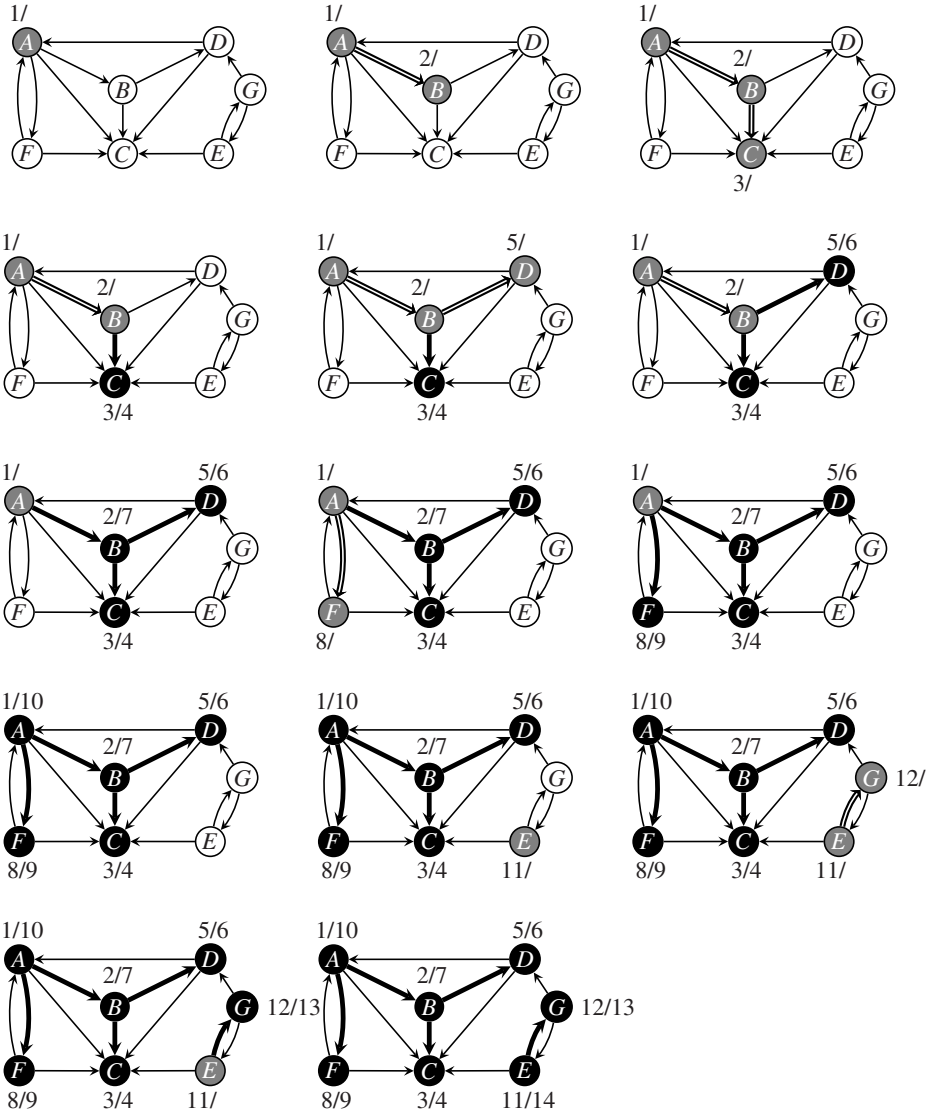


Figura 7.16 Avance de rastreo DFS por el grafo del ejemplo 7.6. Las líneas dobles son aristas de árbol sobre las que todavía no se ha retrocedido, así que conducen a vértices grises. Las líneas gruesas son aristas de árbol por las que ya se retrocedió, así que conducen a vértices negros. Los pares *d*/*f* designan los tiempos de descubrimiento y de finalización de los vértices. Se construyen dos árboles de búsqueda primero en profundidad. Un orden de vértices distinto puede producir un rastreo distinto.

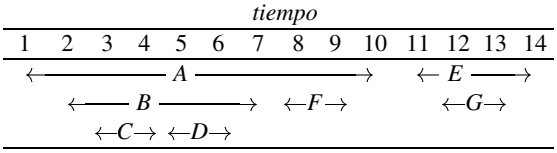


Figura 7.17 Intervalos *activo* para el bosque de búsqueda primero en profundidad de la figura 7.16

Demostración Desglosemos el inciso 1 en el inciso 1(si) y el inciso 1(sólo si), donde el inciso 1(si) es la declaración "si w es un descendiente de v en el bosque DFS, entonces *activo*(w) $\subseteq$ *activo*(v)", y el inciso 1(sólo si) es la contraria. Definamos un orden parcial en V con la regla de que $w < v$ si y sólo si w es un descendiente propio de v en su árbol DFS. Primero demostramos el inciso 1(si) por inducción con este orden parcial. Los casos base son los vértices v que son mínimos en el orden parcial, es decir, vértices sin descendientes propios. Puesto que v es descendiente de sí mismo, el inciso 1(si) se cumple.

Si v no es un vértice mínimo, suponemos que el inciso 1(si) se cumple para todo $x < v$. Si w es cualquier descendiente propio de v en el árbol DFS, existe algún x tal que $vx \in E$ es una arista de árbol dentro del camino que conduce a w, así que w es un descendiente de x. Por inspección de `rastreoDfs` vemos que *activo*(x) $\subset$ *activo*(v). Por la hipótesis inductiva, *activo*(w) $\subset$ *activo*(x). Entonces, *activo*(w) $\subset$ *activo*(v). Con esto se demuestra el inciso 1(si).

Ahora consideramos el inciso 2. Es evidente que esto se cumple si v y w están en árboles DFS distintos, pues todos los vértices de un árbol se procesan antes que cualquier vértice del otro árbol. Supóngase que v y w están en el mismo árbol DFS (pero no tienen relación antepasado/descendiente). Entonces existe un tercer vértice c, su "mínimo común antepasado", tal que existen caminos en el árbol de c a v y de c a w, y estos caminos no tienen aristas en común (véase el ejercicio 7.14). Supóngase que la primera arista del camino de c a v es cy y que la primera arista del camino de c a w es cz. Por inspección de `rastreoDfs` vemos que *activo*(y) y *activo*(z) son intervalos disjuntos. Sin embargo, por el inciso 1(si), *activo*(v) está contenido en *activo*(y) y *activo*(w) está contenido en *activo*(z), así que *activo*(v) y *activo*(w) son también intervalos disjuntos, con lo que terminamos la demostración del inciso 2.

Volvamos ahora al inciso 1(sólo si). Si w *no* es un descendiente de v, entonces w es un antepasado propio de v o bien no existe relación antepasado/descendiente. Si w es un antepasado propio, entonces 1(si) demostró que *activo*(w) $\supset$ *activo*(v), así que *activo*(w) $\not\subset$ *activo*(v), y el inciso 1(sólo si) se cumple para este caso. Si no hay relación antepasado/descendiente, entonces el inciso 2 implica el inciso 1(sólo si).

La demostración del inciso 3 se deja como ejercicio. □

Corolario 7.2 Los vértices que se descubren mientras v está activo son exactamente los descendientes de v en su árbol de búsqueda primero en profundidad. □

Hemos visto mediante un ejemplo que una búsqueda primero en profundidad descubre todos los vértices a los que puede llegarse por un camino de vértices *no descubiertos*. El ejercicio 7.13 pide un ejemplo en el que algunos vértices sean accesibles desde v y no se hayan descubierto aun

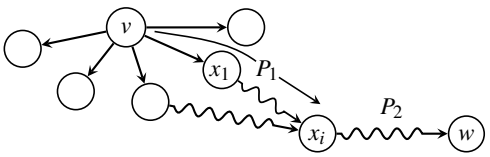
cuando se inicia la búsqueda primero en profundidad de v, pero no se descubran ni visiten mientras v está activo. El teorema que sigue caracteriza exactamente los vértices que *sí* se descubrirán mientras v está activo.

Teorema 7.3 (Teorema del camino blanco) En cualquier búsqueda primero en profundidad de un grafo G, un vértice w es un descendiente de un vértice v en un árbol de búsqueda primero en profundidad si y sólo si, en el momento en que se descubre el vértice v (justo antes de colorearlo gris), existe en G un camino de v a w que consta exclusivamente de vértices blancos.

Demostración (Sólo si) Si w es un descendiente de v, por el teorema 7.1 el camino de aristas de árbol que va de v a w es un camino blanco.

(Si) La demostración es por inducción con k, la longitud de un camino blanco de v a w. El caso base es $k = 0$; entonces $v = w$ y el teorema se cumple.

Para $k \geq 1$, sea $P = (v, x_1, \ldots, x_k)$, donde $x_k = w$, un camino blanco de longitud k que va de v a w. Sea ahora x_i el vértice distinto de v en este camino que se descubre *antes que ningún otro* durante el intervalo *activo* de v. En el diagrama que sigue, las líneas onduladas son caminos, que podrían estar vacíos.



Afirmamos que x_i debe existir, porque x_1 es blanco y tiene una arista directa que viene de v, así que por lo menos se descubre x_1 durante el intervalo *activo* de v. Dividimos el camino P en P_1 de v a x_i, y P_2 de x_i a w (podría ser que $x_i = w$). Sin embargo, P_2 tiene menos de k aristas, y en el momento en que se descubre x_i, P_2 es un camino blanco. Por tanto, por la hipótesis inductiva, w es un descendiente de x_i. Sin embargo,

$$\texttt{tiempoDescubrir}[v] < \texttt{tiempoDescubrir}[x_i] < \texttt{tiempoTerminar}[v]$$

así que por el teorema 7.1 x_i es un descendiente de v. Por transitividad, w es un descendiente de v. □

7.4.6 Grafos acíclicos dirigidos

Los *grafos acíclicos dirigidos* (*DAG*, por sus siglas en inglés) son un caso especial importante de los grafos dirigidos generales. Como su nombre implica, un DAG es cualquier grafo dirigido que no tiene ciclos. Los grafos acíclicos dirigidos son importantes por dos razones primordiales:

1. Muchos problemas se plantean naturalmente en términos de un DAG, como los problemas de calendarización. En esos problemas suele ser necesario que ciertas tareas se finalicen para que otras puedan iniciarse. Un ciclo en las dependencias de las tareas implicaría un *bloqueo mortal*: ninguna tarea del ciclo podrá jamás estar lista para iniciarse.

2. Muchos problemas de grafos dirigidos generales se resuelven con más facilidad —o sea, con mayor eficiencia— en un DAG. La diferencia puede llegar a ser la de tiempo exponencial contra tiempo lineal. Mencionaremos tales problemas cuando se nos presenten en sus versiones generales.

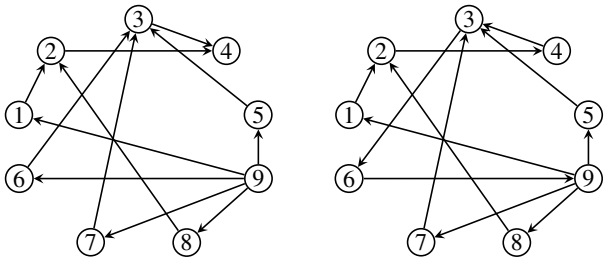


Figura 7.18 Dos grafos dirigidos. ¿Cuál es acíclico?

Además, en la sección 7.5 veremos que todo grafo dirigido general está asociado a cierto grafo acíclico dirigido: su *grafo de condensación*.

Un grafo acíclico dirigido corresponde matemáticamente a un orden parcial de sus vértices. Siempre que existe una arista vw, podemos interpretarla como la relación $v < w$ entre los vértices. Si existe cualquier camino dirigido de v a w, también lo interpretamos como $v < w$ por transitividad. (Todas las aristas de un grafo se podrían interpretar también como $v > w$; sólo es necesario mantener la consistencia dentro del grafo.) Una relación de orden (o de orden parcial) no puede contener ciclos. Veremos que la interpretación de orden es útil en los problemas de calendarización.

En esta sección estudiaremos dos aplicaciones de los DAG: orden topológico y rutas críticas.

Orden topológico

Al meditar acerca de algún problema de grafos dirigidos, el lector podría preguntarse: "Si pudiera dibujar este grafo de modo que todas las aristas apuntaran en general de izquierda a derecha, ¿ello me ayudaría a resolver el problema?" Claro que, si el grafo tiene un ciclo, obviamente sería imposible hacerlo. Pero si el grafo dirigido no tiene ciclos —si es un DAG— veremos que *sí* es posible acomodar los vértices de esa manera. Encontrar tal acomodo es el problema del *ordenamiento topológico*.

Definición 7.16 Orden topológico

Sea $G = (V, E)$ un grafo dirigido con n vértices. Un *orden topológico* para G es una asignación de enteros distintos $1, \ldots, n$ a los vértices de V (sus *números topológicos*) tal que, para cada arista $vw \in E$, el número topológico de v es menor que el número topológico de w. Un *orden topológico inverso* es similar, sólo que, para cada arista $vw \in E$, el número topológico de v es mayor que el número topológico de w. ■

La figura 7.18 muestra dos grafos, sólo uno de los cuales es acíclico. Invitamos al lector a tratar de determinar, por prueba y error, cuál grafo es acíclico, y a tratar de hallar un orden topológico para ese grafo. Un poco de experimentación lo convencerá de que intentar esto sin un plan con un grafo de 50 a 100 vértices no sería factible. Veremos que este problema se puede resolver de manera muy eficiente basándose en el esqueleto de búsqueda primero en profundidad. (En los ejercicios se menciona otra solución eficiente.)

Observemos que un orden topológico equivale a una permutación de los vértices. La definición no especifica que G tenga que ser acíclico, pero es fácil demostrar el lema siguiente.

Lema 7.4 Si un grafo dirigido G tiene un ciclo, G no tiene un orden topológico. □

En cierto sentido, el ordenamiento topológico es el problema fundamental de los DAG. Veremos que todo DAG tiene al menos un orden topológico, demostrando así la contraria del lema 7.4. Una vez hallado un orden topológico para los vértices, muchos otros problemas se vuelven sencillos. El concepto de orden topológico por sí solo podría bastar para sugerir una solución eficiente sin asignar explícitamente los números topológicos.

Ejemplo 7.14 Calendarización sin dependencias de tareas

Consideremos el problema de calendarizar un proyecto que consiste en un conjunto de tareas interdependientes que una persona debe efectuar. Ciertas tareas *dependen* de otras; es decir, no pueden iniciarse en tanto no se hayan llevado a cabo todas las tareas de las que dependen. La forma más natural de organizar la información para un problema de este tipo es como un arreglo de tareas, cada una con una lista de tareas de las que depende directamente. He aquí un ejemplo para el "proyecto" de salir de la casa en la mañana. Las tareas se han numerado en orden aleatorio.

Tarea y número		Depende de
escoger ropa	1	9
vestirse	2	1, 8
desayunar	3	5, 6, 7
salir	4	2, 3
preparar café	5	9
tostar pan	6	9
hacer jugo	7	9
ducharse	8	9
despertar	9	—

Si adoptamos la convención de que vw significa que w depende directamente de v, es decir, que las aristas van "hacia adelante en el tiempo", veremos que la tabla anterior nos da listas de aristas que *entran* en cada vértice. Uno de los grafos de la figura 7.18 corresponde a este conjunto de tareas y dependencias con aristas que apuntan "hacia adelante en el tiempo". En cambio, si interpretamos las listas de dependencias de la tabla como listas de adyacencia para el grafo, obtenemos la transpuesta del grafo "hacia adelante en el tiempo". Es muy común en problemas de calendarización utilizar este grafo transpuesto, también llamado *grafo de dependencia* o *grafo de precedencia*, en el que las aristas apuntan "hacia atrás en el tiempo".

Existen numerosos órdenes topológicos para el conjunto de tareas de esta tabla. Buscaremos uno después de haber presentado el algoritmo para el orden topológico. ■

Los algoritmos para el orden topológico y el orden topológico inverso son simples modificaciones del esqueleto de DFS. Daremos la versión para el orden topológico inverso porque es el que surge más a menudo.

Algoritmo 7.5 Ordenamiento topológico inverso

Entradas: Las mismas que para el esqueleto DFS (algoritmo 7.3) más un arreglo global `topo` y un contador global, `numTopo`.

Salidas: El algoritmo llena el arreglo global `topo` con un conjunto de números topológicos inversos. Los tipos devueltos por este algoritmo se pueden cambiar a **`void`**.

Comentario: Para calcular el orden topológico "hacia adelante", se asigna a `numTopo` el valor inicial $n + 1$ y se cuenta hacia atrás.

Estrategia: Se modifica el esqueleto DFS del algoritmo 7.3 como sigue:

1. En `barridoDfs`, se asigna a `numTopo` el valor inicial 0.
2. En el procesamiento en orden posterior (línea 13) del esqueleto, se inserta

```
numTopo ++; topo[v] = numTopo;   ■
```

Al comparar los algoritmos 7.4 y 7.5, es evidente que ordenar los vértices por sus tiempos de terminación en la búsqueda primero en profundidad produce el mismo ordenamiento que el algoritmo 7.5; el último vértice que termina tiene el número más grande.

Por su conexión con el esqueleto DFS, queda claro que el algoritmo 7.5 se ejecuta en tiempo $\Theta(n + m)$ con un grafo de n vértices y m aristas. Su corrección se demuestra con el teorema siguiente.

Teorema 7.5 Si G es una DAG con n vértices, el algoritmo 7.5 calcula un orden topológico inverso para G en el arreglo `topo`. Por tanto, todo DAG tiene un orden topológico inverso y un orden topológico.

Demostración Puesto que la búsqueda primero en profundidad visita cada vértice exactamente una vez, el código que se insertó en la línea 13 se ejecuta exactamente n veces, así que los números almacenados en el arreglo `topo` son enteros distintos dentro del intervalo $1, \ldots, n$. Sólo falta comprobar que, para cualquier arista vw, `topo[v] > topo[w]`. Consideremos las posibles clasificaciones de vw según la definición 7.14. Si vw fuera una arista de retorno, completaría un ciclo y G no sería un DAG. Para los demás tipos de aristas, en el momento en que se asigna un valor a `topo[v]`, el vértice w se termina (se colorea negro), así que ya se asignó antes un valor a `topo[w]`. Puesto que `numTopo` es siempre creciente, `topo[v] > topo[w]`. □

Ejemplo 7.15 Orden topológico inverso para grafo de dependencia

Hay muchos órdenes topológicos inversos para el grafo de dependencia del ejemplo 7.14. El que halla el algoritmo 7.5 utilizando los números de vértices y las listas de aristas de ese ejemplo es el siguiente:

9	1	8	2	5	6	7	3	4
despertar	escoger ropa	ducharse	vestirse	preparar café	tostar pan	hacer jugo	desayunar	salir

Los resultados de `rastreoDfs` con este grafo se muestran en la figura 7.19. Obsérvese que este grafo es el grafo transpuesto de uno de los grafos de la figura 7.18. ■

Análisis de ruta crítica

El análisis de ruta crítica tiene que ver con hallar un orden topológico, pero es un problema de optimización en el sentido de que se debe descubrir el camino más largo dentro del DAG. Al igual que con el problema de calendarización del ejemplo 7.14, un proyecto consta de un conjunto de tareas, y las tareas tienen dependencias. Ahora, empero, suponemos que también se nos da el tiempo necesario para llevar a cabo cada tarea, una vez iniciada. Además, suponemos que todas las tareas que están listas para efectuarse *se pueden* efectuar simultáneamente; es decir, hay suficientes trabajadores para asignar un trabajador distinto a cada tarea. Ciertamente, el último supuesto es dudoso en muchas situaciones prácticas, pero para poner manos a la obra haremos este supuesto simplificador.

Podemos definir el *tiempo de terminación mínimo* de una tarea, suponiendo que el proyecto se inicia en el tiempo 0, como sigue.

Definición 7.17 Tiempos de inicio y terminación mínimos, ruta crítica

Un *proyecto* consta de un conjunto de *tareas*, numeradas $1, \ldots, n$. Cada tarea tiene una lista de *dependencias*, que son las tareas de las que depende directamente, y un número no negativo que denota su *duración*. El *tiempo de inicio mínimo* (`tim`) de una tarea v es

1. cero si v no tiene dependencias,
2. el máximo de los tiempos de terminación mínimos (véase más adelante) de sus dependencias, si v tiene dependencias.

El *tiempo de terminación mínimo* (`ttm`) de cualquier tarea es su tiempo de inicio mínimo más su duración.

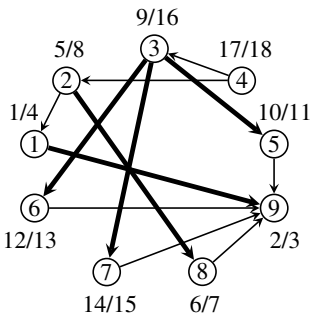


Figura 7.19 Resultados de `rastreoDfs` con el grafo de dependencia del ejemplo 7.14. Las líneas gruesas son aristas de árbol, y los pares *d*/*f* designan los tiempos de descubrimiento y terminación de los vértices. Obsérvese que hay cuatro árboles de búsqueda primero en profundidad. (¿Dónde está el cuarto?)

Una *ruta crítica* en un proyecto es una sucesión de tareas $v_0, v_1, \ldots, v_k$ tal que

1. v_0 no tiene dependencias;
2. para cada tarea subsiguiente, v_i $(1 \leq i \leq k)$, v_{i-1} es una dependencia de v_i tal que `tim` de v_i es igual a `ttm` de v_{i-1}, y
3. `ttm` de v_k es máximo para todas las tareas del proyecto. ■

Una ruta crítica no tiene "holganza"; es decir, no se hace pausa alguna entre la terminación de una tarea de la ruta y el inicio de la siguiente. En otras palabras, si v_i sigue a v_{i-1} en una ruta crítica, el `ttm` de v_{i-1} deberá haber sido máximo entre todas las dependencias de v_i. Por tanto, v_{i-1} es una *dependencia crítica* de v_i, en el sentido de que cualquier retraso en v_{i-1} causará un retraso en v_i. Desde otro punto de vista, supóngase que estamos buscando una manera de acelerar la terminación de todo el conjunto de tareas buscando una forma más rápida de llevar a cabo una de ellas. Es evidente que reducir el tiempo de una tarea no ayuda a reducir el tiempo total requerido si la tarea no está en una ruta crítica. El interés práctico en las rutas críticas se basa en estas propiedades.

Para no complicar demasiado nuestro problema, hemos supuesto que cada tarea tiene una duración fija. En muchas situaciones reales, la duración se puede acortar asignando más recursos a la tarea, tal vez tomando algunos de ellos de una tarea que no está en la ruta crítica.

Ejemplo 7.16 Ruta crítica

Conservemos las tareas y dependencias del ejemplo 7.14 y agreguemos duraciones (en minutos):

9	1	8	2	5	6	7	3	4
despertar	escoger ropa	ducharse	vestirse	preparar café	tostar pan	hacer jugo	desayunar	salir
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Así pues, realizar las tareas una tras otra toma 32.0 minutos. Supóngase que podemos hacerlas todas simultáneamente, restringidos sólo por el requisito de terminar las dependencias antes de iniciar una tarea. La ruta crítica va de despertar a ducharse a vestirse a salir: ¡sólo 16 minutos! Para lograr esto, es preciso traslapar las actividades de hacer el jugo, desayunar y vestirse (lo cual no es demasiado descabellado), pero también es preciso escoger la ropa, tostar el pan y preparar café mientras nos duchamos (lo cual es un poco más complicado). ■

Hemos dado las definiciones naturales en términos de tareas y duraciones. Se requiere un poco de manipulación para relacionar estos términos con caminos más largos, porque no hemos definido la longitud de las aristas. Además, el número de aristas en un camino es uno menos que el número de tareas, así que ¿cómo podría una longitud de camino tomar en cuenta la duración de *todas* las tareas del camino? Podemos precisar la conexión con unas cuantas modificaciones técnicas:

1. Añadimos una tarea especial al proyecto, llamada *hecho*, con duración 0; puede ser la tarea número $n + 1$.
2. Hacemos que toda tarea normal que no sea dependencia de otra tarea (es decir, que pudiera ser una tarea final) sea una dependencia de *hecho*.

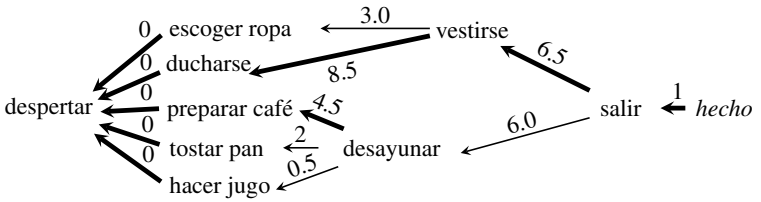
3. El *DAG del proyecto* tiene una arista ponderada vw en todos los casos en que v depende de w, y el peso de dicha arista es la duración de w.

Cabe señalar que hemos optado por que las aristas apunten de la tarea a su tarea dependiente, por concordancia con la forma en que suele organizarse la información de dependencia; ésta es "hacia atrás en el tiempo". Si es más conveniente que las aristas apunten "hacia adelante en el tiempo", también puede hacerse.

Ahora vemos que un camino más largo en el DAG del proyecto corresponde a una ruta crítica según la definición original; simplemente tiene el vértice adicional *hecho* al principio. La distancia de la tarea *hecho* a cualquier tarea v de este camino es la diferencia entre el tiempo de inicio mínimo de *hecho* y el de v. Esta distancia es máxima cuando el tiempo de inicio mínimo de v es 0. Es así como podemos usar un algoritmo para calcular caminos más largos en un DAG para hallar rutas críticas.

Ejemplo 7.17 Ruta crítica como camino más largo

Se muestra el grafo ponderado con el vértice *hecho* para el problema de ruta crítica del ejemplo 7.16; las líneas gruesas identifican la ruta crítica y las subrutas críticas.



Una subruta crítica es el camino más largo que sale de un vértice, no necesariamente el vértice *hecho*. Por ejemplo, no podemos comenzar a desayunar si el café no está preparado; otras preparaciones terminan antes. ■

Al igual que el algoritmo para el orden topológico inverso, un algoritmo para rutas críticas se basa en gran parte en el esqueleto DFS.

Algoritmo 7.6 Ruta crítica

Entradas: Las mismas que recibe el esqueleto DFS (algoritmo 7.3) más los arreglos globales `duracion`, `depCrit` y `ttm`. Una condición previa es que G sea un DAG. Las aristas de G apuntan de las tareas a sus dependencias (hacia atrás en el tiempo).

Salidas: El algoritmo llena los arreglos globales `depCrit` y `ttm`; `ttm[v]` es el tiempo de terminación mínimo de v y `depCrit[v]` es una dependencia crítica de v. Se puede hallar una ruta crítica rastreando hacia atrás desde un vértice con valor `ttm` máximo, siguiendo los valores `depCrit` como ligas. Los tipos devueltos por este algoritmo se pueden cambiar a **`void`**.

Comentario: El algoritmo sólo requiere ajustes menores para funcionar si las aristas apuntan "hacia adelante en el tiempo".

Estrategia: En el procedimiento `dfs` recursivo, una variable local `tim` contendrá el tiempo de inicio mínimo, que es el máximo de los valores `ttm` de las dependencias de la tarea actual. Modificamos el esqueleto DFS del algoritmo 7.3 como sigue:

1. En el procesamiento en orden previo (línea 2) del esqueleto, insertamos

```
tim = 0; depCrit[v] = -1;
```

2. En el punto de retroceso tanto por aristas de árbol (línea 9) como por aristas no de árbol (línea 11) del esqueleto, insertamos

```
if (ttm[w] >= tim)
    tim = ttm[w];
    depCrit[v] = w;
```

   Es importante recordar aquí que la arista no de árbol no puede ser una arista de retroceso, pues en tal caso no se habría asignado ningún valor inicial a `ttm[w]`.

3. En el procesamiento en orden posterior (línea 13) del esqueleto, insertamos

```
ttm[v] = tim + duracion[v];
```

■

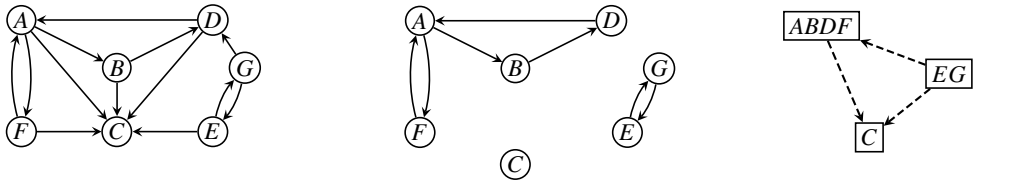
Una vez más, es evidente que el algoritmo se ejecuta en tiempo $\Theta(n + m)$ con n vértices y m aristas. Por la naturaleza de la búsqueda primero en profundidad, se asigna un valor a cada elemento del arreglo `ttm` exactamente una vez, para los índices $1, \ldots, n$. Por tanto, el código insertado meramente implementa las definiciones de `ttm` y `tim`, siempre que los valores de `ttm[w]` a los que se accede estén definidos en el momento en que se accede a ellos. No obstante, esto es consecuencia del argumento que usamos al demostrar el teorema 7.5, de que w ya está terminado (se coloreó negro) en dicho momento.

Resumen de grafos acíclicos dirigidos

Hemos visto que los grafos acíclicos dirigidos se usan en problemas de calendarización, por lo que el problema del orden topológico y el de la ruta crítica se pueden resolver insertando unas cuantas líneas de código en el esqueleto DFS. Los DAG tienen muchas otras aplicaciones, y apenas hemos rascado la superficie de este tema. En los ejercicios se presentan algunos problemas adicionales. En la sección siguiente veremos que *godo* grafo dirigido está asociado a cierto DAG: su *grafo de condensación*. Así, las aplicaciones de los DAG podrían extenderse en algunos casos a las aplicaciones de grafos con ciclos.

7.5 Componentes fuertemente conectados de un grafo dirigido

Un grafo no dirigido está conectado si y sólo si existe un camino entre cada par de vértices. La conectividad de grafos dirigidos se puede definir por una de dos maneras, dependiendo de si exigimos o no que las aristas se recorran únicamente de su cabeza a su cola. Recordemos la definición 7.6, que dice que un grafo dirigido $G = (V, E)$ está *fuertemente conectado* si, para cada par de vértices v y w, existe un camino de v a w (y por ende, intercambiando los papeles de v y w en la definición, existe también un camino de w a v). Es decir, las aristas se deben seguir en la dirección de su "flecha". G está *débilmente conectado* si, después de hacer que las aristas no estén dirigidas y consolidar cualesquier aristas repetidas, el grafo no dirigido resultante está conectado. Nos concentraremos en la conectividad fuerte.



(a) El grafo dirigido (b) Sus componentes fuertes (c) Su grafo de condensación

Figura 7.20 Componentes fuertes y grafo de condensación del grafo dirigido empleado en la figura 7.16 y en varios ejemplos.

Definición 7.18 Componente fuertemente conectado

Un *componente fuertemente conectado* (que en adelante llamaremos *componente fuerte*) de un grafo dirigido G es un subgrafo máximo fuertemente conectado de G. (El significado de "máximo" se explicó después de la definición 7.8.) ■

Podríamos dar una definición alterna en términos de una relación de equivalencia, S, entre los vértices. Para v y w en V, sea vSw si y sólo si existe un camino de v a w y un camino de w a v. (Recordemos que vSw es otra forma de escribir $(v, w) \in S$, donde $S \subseteq V \times V$. Aquí, (v, w) es cualquier par ordenado de vértices, no necesariamente una arista de G. Las relaciones de equivalencia se definieron en la sección 1.3.1.) Entonces, un componente fuerte consiste en una clase de equivalencia, C, junto con todas las aristas vw tales que v y w están en C. Véase el ejemplo de la figura 7.20. A veces usamos el término *componente fuerte* para referirnos únicamente al conjunto de vértices C; el significado deberá quedar claro por el contexto.

Cada uno de los componentes fuertes de un grafo dirigido se puede reducir a un solo vértice para dar un nuevo grafo dirigido que no tiene ciclos.

Definición 7.19 Grafo de condensación

Sean $S_1, S_2, \ldots, S_p$ los componentes fuertes de G. El *grafo de condensación de G* (o simplemente *condensación de G*), denotado por $G\downarrow$, es el grafo dirigido $G\downarrow = (V', E')$, donde V' tiene p elementos, $s_1, s_2, \ldots, s_p$, y $s_i s_j$ está en E' si y sólo si $i \neq j$ y existe una arista en E que va de algún vértice que está en S_i a algún vértice que está en S_j. Dicho de otro modo, todos los vértices de S_i se condensan en un solo vértice s_i. ■

En la figura 7.20 se muestra un ejemplo. En ejemplos pequeños usaremos la convención de que el nombre de un vértice condensado es simplemente la concatenación de los nombres de todos los vértices del componente fuerte. Obsérvese que las aristas originales AC, BC, DC y FC se han reducido a una arista.

Las soluciones a algunos problemas de grafos dirigidos se pueden simplificar tratando por separado los componentes fuertes y la condensación, aprovechando las propiedades especiales de cada uno: los primeros están fuertemente conectados y la segunda es acíclica. (Por ejemplo, con-

sideremos la relación entre los componentes fuertes y la condensación del diagrama de flujo de un programa a la estructura cíclica del programa.)

7.5.1 Propiedades de los componentes fuertemente conectados

Los componentes fuertes tienen varias propiedades interesantes, que veremos a continuación. El algoritmo para determinar los componentes fuertes se presenta en la sección 7.5.2, y su funcionamiento se puede entender sin leer esta subsección. No obstante, este material es importante para entender *por qué* funciona.

Recordemos la definición 7.10, según la cual G^T, el grafo transpuesto de G, es el resultado de invertir la dirección de cada arista de G. Se sigue inmediatamente de la definición que los componentes fuertes de G^T son idénticos, en términos de vértices, a los componentes fuertes de G. Las aristas también son idénticas, excepto por su dirección. Además, $(G\downarrow)^T = (G^T)\downarrow$; es decir, la condensación de G es igual a la condensación de G^T, con la salvedad de que la dirección de las aristas está invertida.

Consideremos ahora la relación entre los árboles de búsqueda primero en profundidad y los componentes fuertes. Veremos que, para fines estructurales, un *líder*, que definiremos a continuación, básicamente representa la totalidad de su componente fuerte.

Definición 7.20 Líder de un componente fuerte

Dado un grafo dirigido G con componentes fuertes S_i, $i = 1, \ldots, p$, y una búsqueda primero en profundidad de G, el primer vértice de S_i que se descubre durante la búsqueda es el *líder* de S_i y se denota con v_i. ■

Supóngase que aún no se ha descubierto ningún vértice de G. Si una búsqueda primero en profundidad inicia en v_1 (es decir, v_1 es la raíz de un árbol de búsqueda primero en profundidad y es el líder del componente fuerte S_1), entonces, por el teorema del camino blanco (teorema 7.3), todos los vértices de S_1 serán descendientes de v_1 en el árbol de búsqueda primero en profundidad. Es más, si es posible llegar a cualquier vértice de otro componente fuerte, digamos S_j, el primero de esos vértices que se descubra será v_j y, aplicando el teorema del camino blanco en el momento en que se descubre v_j vemos que se descubre la totalidad de S_j en este árbol. Lo mismo sucede con los árboles de búsqueda primero en profundidad subsiguientes. Esto demuestra el lema siguiente.

Lema 7.6 Cada árbol de búsqueda primero en profundidad de un bosque de búsqueda primero en profundidad de un grafo dirigido G contiene uno o más componentes fuertes completos de G. No hay componentes fuertes "parciales" en ningún árbol de búsqueda primero en profundidad. □

Corolario 7.7 El líder v_i es el último vértice en terminar (es decir, en llegar al procesamiento en orden posterior y ser coloreado de negro) de todos los vértices de S_i. □

¿Hay alguna forma de organizar el orden de búsqueda de modo que un árbol contenga *exactamente un* componente fuerte? Para obtener una pista, examinemos otra vez la figura 7.20(c). El grafo de condensación muestra claramente que si iniciamos nuestra búsqueda primero en profundidad en cualquier parte de un componente fuerte del cual no *salgan* flechas, esa búsqueda deberá descubrir exactamente un componente fuerte. El subgrafo C cumple con este requisito. Pero,

¿cómo nos puede ayudar esto en la práctica? ¡No *conocemos* el grafo de condensación antes de hallar los componentes fuertes! El secreto se revelará en la próxima subsección, pero su corrección depende de otras propiedades de los líderes.

Aunque no conocemos los componentes fuertes ni los líderes, podemos sacar algunas conclusiones. Supóngase que sale una arista de un componente fuerte S_i y entra en otro, S_j. Esto implica que existe un camino de v_i a v_j en G. Desde luego, es posible que v_j sea un descendiente de v_i en el árbol de búsqueda primero en profundidad que contiene a v_i. En este caso, $activo(v_j) \subset activo(v_i)$. ¿Qué otras posibilidades hay? Es obvio que v_j no puede ser antepasado de v_i, pues si lo fuera estarían en el mismo componente fuerte. En el caso de vértices generales, el intervalo *activo* de v_j podría estar en su totalidad antes del de v_i o en su totalidad después (y también contenido en él). Demostraremos que, en el caso de líderes conectados por un camino, se puede descartar una de estas posibilidades. Invitamos al lector a tratar de determinar cuál antes de continuar.

■ ■ ■

Lema 7.8 En el momento en que se descubre un líder v_i durante una búsqueda primero en profundidad (justo antes de que se le pinte de gris), no existe ningún camino entre v_i y algún vértice gris, digamos x.

Demostración En este momento, todo vértice gris es un antepasado propio de v_i y se le descubrió antes que a v_i, así que debe estar en un componente fuerte distinto. Puesto que existe un camino de x a v_i, no debe haber un camino de v_i a x. □

Lema 7.9 Si v es el líder de su componente fuerte S, y x está en un componente fuerte distinto, y existe un camino de v a x en G, entonces en el momento en que v es descubierto x es negro o bien existe un camino blanco de v a x (y x es blanco). En ambos casos, v termina después de x.

Demostración Consideremos cualquier camino de v a x, y consideremos el último vértice *no* blanco, digamos z, de ese camino. Si no existe z, quiere decir que todo el camino es blanco y ya terminamos. Supóngase que z existe. Por el lema 7.8, z debe ser negro. Si $z = x$, ya terminamos. Supóngase $z \neq x$, pero ahora consideremos el momento (anterior) en el que se descubrió z. El camino de z a x era blanco en ese momento, así que por el teorema del camino blanco x es un descendiente de z y ahora también es negro. □

Por tanto, si existe una arista de S_i a S_j, hemos descartado la posibilidad de que el intervalo *activo* de v_j, el líder de S_j, esté en su totalidad después del de v_i, el líder de S_i. Recomendamos al lector demostrar mediante ejemplos que el lema no se cumpliría (y $activo(v_j)$ podría estar totalmente después de $activo(v_i)$) si no exigiéramos que v_i fuera el líder de su componente fuerte. El ejercicio 7.13 tiene que ver con esto.

7.5.2 Un algoritmo para componentes fuertes

Estudiaremos un algoritmo para hallar componentes fuertes que aprovecha la mayor parte de las propiedades que vimos en la sección anterior. El primer algoritmo para componentes lineales que se ejecuta en tiempo lineal se debe a R. E. Tarjan, y se basa en la búsqueda primero en profundidad. El algoritmo que presentaremos se debe a M. Sharir, y también se basa en la búsqueda primero en profundidad. Es elegante por su sencillez y sutileza.

El algoritmo tiene dos fases principales:

1. Se efectúa una búsqueda primero en profundidad estándar en G, y los vértices se colocan en una pila en el momento en que se termina con ellos.
2. Se efectúa una búsqueda primero en profundidad en G^T, el grafo transpuesto, aunque se emplea un método poco usual para hallar vértices blancos desde los cuales iniciar una búsqueda (es decir, un nuevo árbol): se sacan vértices de la pila que se construyó durante la fase 1, en lugar de que un ciclo **for** acceda a ellos en orden numérico (como en `barridoDfs` del algoritmo 7.3, que se usó en la fase 1). Durante esta búsqueda, el algoritmo almacena el líder del componente fuerte de cada vértice v en `cfc[v]`.

Cada árbol de búsqueda primero en profundidad generado en la fase 2 será exactamente un componente fuerte. Puesto que los componentes fuertes se hallan en la fase 2, en realidad estamos hallando los componentes fuertes de G^T. No obstante, como ya señalamos, los componentes fuertes de G^T y de G son idénticos en términos de sus vértices, y sus aristas coinciden con la salvedad de que apuntan en la dirección opuesta.

Algoritmo 7.7 Componentes fuertemente conectados

Entradas: Un arreglo `verticesAdya` de listas de adyacencia que representa un grafo dirigido $G = (V, E)$, como se describió en la sección 7.2.3, y n, el número de vértices. El arreglo está definido para los índices $1, \ldots, n$; el elemento número 0 no se usa.

Salidas: Un arreglo `cfc` en el que cada vértice se numera para indicar en qué componente fuerte está. El identificador de cada componente fuerte es el número de algún vértice miembro de ese componente (aunque se podrían usar otros sistemas de identificación). (El invocador reserva espacio para `cfc` y lo pasa como parámetro a este procedimiento, el cual lo llena.)

Comentario: El grafo transpuesto G^T podría ser una entrada, en lugar de calcularse en el procedimiento. Cabe señalar que los parámetros tercero y cuarto de `dfsT` son ambos v en la invocación de nivel más alto, pero tienen diferente significado. El tercer parámetro designa el vértice actual que se visitará y cambia en cada invocación recursiva. El cuarto parámetro designa el identificador del componente fuerte y no cambia durante las invocaciones recursivas. Se usan las operaciones del TDA Pila que describimos en la sección 2.4.1; omitimos los calificadores de nombre de clase para hacer más comprensible el código.

```
void componentesFuertes(ListaInt[] verticesAdya, int n, int[] cfc)
// Fase 1
1.  PilaInt pilaTerminar = crear(n);
2.  Efectuar una búsqueda primero en profundidad de G, utilizando el esqueleto DFS del
    algoritmo 7.3. En el procesamiento en orden posterior para el vértice v (línea 13 del es-
    queleto), se inserta el enunciado: push(pilaTerminar, v);
// Fase 2
3.  Calcular G^T, el grafo transpuesto, representado en el arreglo transAdya de listas de ad-
    yacencia.
4.  barridoDfsT(transAdya, n, pilaTerminar, cfc);
    return;
```

```
void barridoDfsT(ListaInt[] transAdya, int n, PilaInt pilaTerminar,
int[] cfc)
// barridoDfs en el grafo transpuesto
    Reservar espacio para el arreglo color e inicializarlo con blanco.
    while (pilaTerminar no esté vacía)
      int v = tope(pilaTerminar);
      pop(pilaTerminar);
      if(color[v] == blanco)
            dfsT(transAdya, color, v, v, cfc);
      // Continuar ciclo.
    return;

void dfsT(ListaInt[] transAdya, int[] color, int v, int lider, int[] cfc)
    Usar el esqueleto de búsqueda primero en profundidad estándar del algoritmo 7.3. En el
    procesamiento en orden previo para el vértice v (línea 2 del esqueleto) insertar el enun-
    ciado:
       cfc[v] = lider;
    Pasar lider y cfc como parámetros de invocaciones recursivas.
```

La pila `pilaTerminar` se puede implementar de forma muy sencilla como un arreglo de n elementos, ya que sólo se efectuarán n operaciones de apilar (`push`) durante la ejecución del algoritmo.

Ejemplo 7.18 Componentes fuertes

Para ver el algoritmo en acción, examinemos el grafo de la figura 7.20(a). La DFS de la fase 1 se detalló en la figura 7.16.

Con la operación `push` que se inserta en el procesamiento en orden posterior, `pilaTerminar` se desarrolla como se muestra en la figura 7.21(a). Antes de continuar, verifiquemos las posiciones relativas de los líderes de los componentes fuertes. (Aunque el algoritmo no las conoce, nosotros podemos conocerlas asomándonos a la figura 7.20 y verificando los tiempos de descubrimiento en la figura 7.16.) Los líderes son A, C y E. Existe un camino de E al resto y E fue el último en terminar. Existe un camino de A a C, y A está más arriba en la pila que C, así que terminó después. Por tanto, se confirma el lema 7.9.

Ahora `componentesFuertes` continúa con las líneas 3 y 4, invocando `barridoDfsT`. El grafo transpuesto se muestra en la figura 7.21(b). En la primera pasada por el ciclo **`while`** de `barridoDfsT`, una `dfsT` se inicia en E, el tope actual de `pilaTerminar`. Obsérvese que E se pasa en el cuarto parámetro como líder y en el tercer parámetro como vértice a visitar.

En el grafo transpuesto, las aristas DG y CE están orientadas *hacia* G y E, respectivamente. Esta `dfsT` queda "atrapada" en un solo componente fuerte, $S_E = \{E, G\}$, y esos vértices constituyen el primer árbol abarcante primero en profundidad de la fase 2, en G^T. (El algoritmo no construye realmente el árbol; esto sólo se hace en nuestro análisis.) Según la descripción de `dfsT`, E se pasa como el parámetro correspondiente al líder en las invocaciones recursivas, y se guarda en el arreglo `cfc` cada vez que se descubre y visita un vértice. Esto nos lleva a la figura 7.21(c).

El control retrocede a `barridoDfsT`, donde se reanuda la búsqueda de otro vértice blanco desapilando `pilaTerminar`. Se desapila el vértice G y se pasa por alto porque es negro a estas alturas. Luego se desapila A y se convierte en la raíz de un nuevo árbol. Demostraremos más ade-

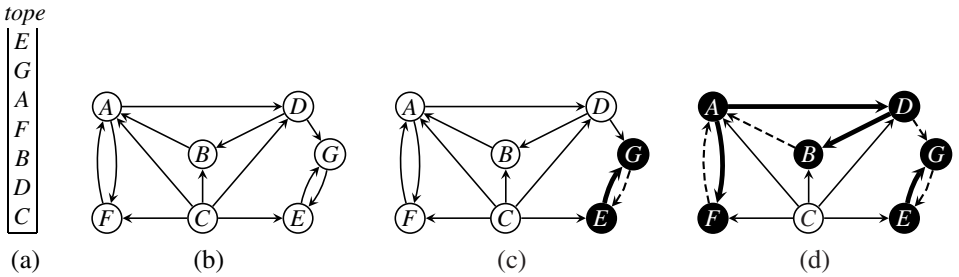


Figura 7.21 Fase 2 del algoritmo Componentes Fuertes: (a) `pilaTerminar` al principio de la fase 2. (b) El grafo transpuesto. (c) El primer árbol DFS identifica un componente fuerte. Las aristas gruesas son aristas de árbol; las aristas punteadas son aristas no de árbol procesadas; las aristas delgadas todavía no se han procesado. (d) El segundo árbol DFS identifica otro componente fuerte. Obsérvese que *DG* es una arista cruzada a un árbol DFS distinto. Obsérvese también que *C* será el último árbol DFS y no tendrá aristas.

lante que todo vértice blanco que se desapila es un líder, como es el caso de *A*. Una vez más, observamos que las aristas van de *C* a *A*, *B*, *D* y *F*, así que esta búsqueda no puede "filtrarse" al componente fuerte de *C*. No obstante, ahora hay una arista de *D* a *G*, que *sí* sale del componente fuerte del cual *A* es líder. Sin embargo, no es coincidencia que *G* ya se haya descubierto antes de iniciarse esta segunda búsqueda y sea actualmente un vértice negro. Por tanto, también se frustra este último intento por "escapar" del componente fuerte de *A*. Esto nos lleva a la figura 7.21(d).

Por último, terminamos el árbol cuya raíz es *C* y que no tiene aristas. Con esto `barridoDfsT` termina su tarea y también termina el algoritmo. Una vez más, recordamos a los lectores que el algoritmo no construye realmente los árboles; esto sólo se hace en nuestro análisis. ■

Al principio parece una coincidencia asombrosa que siempre que una búsqueda primero en profundidad en el grafo transpuesto podría "descarriarse", saliéndose del componente fuerte de su raíz, da la casualidad que el vértice "descarriado" ya se había descubierto en un árbol anterior. No obstante, después de probar varios órdenes de los vértices (en la fase 1) de la figura 7.20, y de probar otros grafos, el lector descubrirá que al parecer las cosas siempre se arreglan. Los lemas siguientes demuestran por qué es así.

Lema 7.10 En la fase 2, cada vez que se desapila un vértice blanco de `pilaTerminar`, ese vértice es el líder en la fase 1 de un componente fuerte.

Demostración El desapilado se efectúa en el orden inverso del tiempo de terminación en la fase 1. Por el corolario 7.7, el líder es el primero vértice de un componente fuerte en desapilarse. Supóngase que se desapila el vértice x y no es un líder. Eso implicaría que algún otro vértice del componente fuerte al que x pertenece fue el primero en visitarse dentro de ese componente fuerte. Por el lema 7.6 y el teorema del camino blanco, x ya está en un árbol terminado, así que x no es blanco. □

Teorema 7.11 En la fase 2, todo árbol de búsqueda primero en profundidad contiene exactamente un componente fuerte de vértices.

Demostración El lema 7.6 dice que todo árbol de búsqueda primero en profundidad contiene uno o más componentes fuertes completos. Así pues, deberemos demostrar que sólo hay uno. Sea v_i

el líder de fase 1 de S_i. Supóngase que v_i se desapila de `pilaTerminar` y es blanco. Entonces, v_i es la raíz de un árbol de búsqueda primero en profundidad en la fase 2. Si no se puede llegar a ningún otro componente fuerte desde v_i, siguiendo un camino en G^T, no habrá problema.

Supóngase que sí se puede llegar a algún otro componente fuerte, digamos S_j cuyo líder es v_j, desde v_i siguiendo un camino en G^T. Entonces existe un camino en G que va de v_j a v_i. Por el lema 7.9, v_j terminó después de v_i en la fase 1, y por ende ya se desapiló de `pilaTerminar` y todos los vértices de S_j son negros en el momento en que se desapila v_i. Por tanto, el árbol de búsqueda primero en profundidad actual no puede "escapar" de S_i. □

Teorema 7.12 El algoritmo `componentesFuertes` identifica correctamente los componentes fuertes de G^T, y son los mismos, en términos de vértices, que los componentes fuertes de G.

Demostración Por el teorema 7.11, cada árbol de búsqueda primero en profundidad contiene exactamente un componente fuerte (de G^T) y, por las propiedades de la búsqueda primero en profundidad, cada uno de los vértices de G^T está en algún árbol de búsqueda primero en profundidad. □

7.5.3 Análisis

Una buena parte del análisis del algoritmo 7.2, componentes conectados, es válido para el algoritmo de componentes fuertes si se hacen algunos cambios menores. El algoritmo realiza dos búsquedas primero en profundidad, cada una de las cuales tarda un tiempo $\Theta(n + m)$. El cálculo de G^T, de ser necesario, también está en $\Theta(n + m)$ (véase el ejercicio 7.8). El espacio extra que ocupan diversos arreglos está en $\Theta(n)$. La pila de recursión también utiliza un espacio $\Theta(n)$ en el peor caso. Así pues, si incluimos las listas de adyacencia de G^T, el espacio utilizado está en $\Theta(n + m)$.

7.6 Búsqueda de primero en profundidad en grafos no dirigidos

La búsqueda primero en profundidad en un grafo no dirigido sigue el mismo tema que en un grafo dirigido: explorar más lejos, si es posible, y retroceder si es necesario. Muchos de los aspectos del esqueleto DFS se pueden usar sin cambios. Podemos emplear el mismo sistema para los colores de los vértices, tiempos de descubrimiento, tiempos de terminación y árboles DFS. No obstante, la búsqueda primero en profundidad en un grafo no dirigido se complica por el hecho de que las aristas se deben *explorar* sólo en una dirección, pero están representadas dos veces en la estructura de datos.

En algunos problemas no importa si una arista se procesa dos veces, como vimos en el problema de los componentes conectados, y el grafo se puede tratar como digrafo simétrico. En esta sección nos ocuparemos de situaciones en las que tal simplificación no funcionaría. Como regla práctica, los problemas en los que intervienen *ciclos* dentro de grafos no dirigidos deben procesar cada arista únicamente una vez. Estudiaremos uno de esos problemas con detalle en la sección 7.7.

En un grafo no dirigido, la búsqueda primero en profundidad imparte una orientación a cada una de sus aristas: están orientadas en la dirección en la que se pasó por ellas inicialmente (se *exploraron*, según la definición presentada al principio de la sección 7.3.1). Incluso si la arista no conduce a un vértice no descubierto, su orientación es tal que *se aleja* del primer vértice que la encontró durante la búsqueda; decimos que ese vértice *verifica* la arista, como en la búsqueda primero en profundidad dirigida. El procesamiento de aristas no de árbol se efectúa cuando se verifican. Las aristas de árbol también están orientadas alejándose del primer vértice que las encontró;

se exploran y posteriormente se retrocede por ellas, igual que en la búsqueda primero en profundidad dirigida.

Cuando un vértice encuentra una arista en la estructura de datos (lista de adyacencia o matriz de adyacencia) que está orientada *hacia* él, pasa por alto esa arista como si no existiera. El esqueleto DFS para grafos dirigidos se modifica de modo que reconozca estas situaciones. Podemos averiguar cómo surgen tales situaciones estudiando las aristas no de árbol en grafos dirigidos simétricos (definición 7.3). Al hacerlo, nos percataremos de lo siguiente:

1. En un grafo dirigido simétrico simplemente no puede haber *aristas cruzadas*.
2. Una *arista de retroceso* de un vértice v a p, su padre en el árbol DFS, sería el segundo encuentro con la arista no dirigida entre esos dos vértices, habiendo sido el primero como la arista de árbol pv; por tanto, es preciso pasar por alto vp. Las demás aristas de retroceso son primeros encuentros.
3. Una *arista delantera* en un grafo dirigido simétrico siempre es el segundo encuentro con la arista no dirigida. Digamos que se encuentra una arista delantera de v a w. Ello querrá decir que w ya se descubrió antes y que wv se procesó en esa orientación como *arista de retorno*. Puesto que no puede haber aristas cruzadas, cualquier arista a un vértice negro deberá ser una arista delantera en el digrafo simétrico y se le deberá pasar por alto en el grafo no dirigido.

Este análisis sugiere las modificaciones siguientes al esqueleto DFS del algoritmo 7.3. Primero, pasamos el padre DFS p como parámetro adicional a `dfs`. Esto permite implementar el inciso 2 de la lista anterior. Luego, al procesar la arista vw, si w no es blanco, probamos si w es gris y si es diferente de p, el padre de v (que se pasó como parámetro). Si lo es, se trata de una arista de retorno "real"; si no, la pasamos por alto por las razones descritas en los incisos 2 y 3. Esta prueba se incorpora a la línea 10 del esqueleto de búsqueda primero en profundidad no dirigida, que daremos a continuación. En el ejercicio 7.28 se pide al lector demostrar que la búsqueda primero en profundidad no dirigida clasifica cada una de las aristas como arista de árbol o arista de retorno.

La rutina `barridoDfs` para grafos no dirigidos sólo difiere de la del algoritmo 7.3 en que la invocación de `dfs` tiene el valor -1 como parámetro de padre. Esto indica que el vértice actual es la raíz de su árbol de búsqueda primero en profundidad. Compárese también con el algoritmo 7.4.

Algoritmo 7.8 Esqueleto de búsqueda primero en profundidad no dirigida

Entradas: Un arreglo `verticesAdya` de listas de adyacencia que representa un grafo no dirigido $G = (V, E)$, según la descripción de la sección 7.2.3, y n, el número de vértices. El arreglo está definido para los índices $1, \ldots, n$. También el arreglo `color` para registrar la situación de búsqueda, el vértice v que será el siguiente en visitarse, y el vértice p que es el padre de v. Los demás parámetros serán los que necesite la aplicación.

Salida: (Efectúa una búsqueda primero en profundidad partiendo del vértice v.) Los parámetros y el valor devuelto `respuesta` serán los que necesite la aplicación. El tipo devuelto **`int`** es sólo un ejemplo. El arreglo `color` también se actualiza de modo que todos los vértices descubiertos durante esta `dfs` sean negros; los demás no cambian.

Comentarios: La envoltura `barridoDfs` es como la del algoritmo 7.3, excepto que invoca a `dfs` con el cuarto parámetro (p) puesto en -1. Los significados de los colores son: blanco = no descubierto, gris = activo, negro = terminado.

```
int dfs(ListaInt[] verticesAdya, int[] color, int v, int p, ...)
   int w;
   ListaInt adyaRest;
   int respuesta;

1.  color[v] = gris;
2.  Procesamiento en orden previo del vértice v
3.  adyaRest = verticesAdya[v];
4.  while (adyaRest ≠ nil)
5.     w = primero(adyaRest);
6.     if (color[w] == blanco)
7.        Procesamiento exploratorio de arista de árbol vw
8.        int wResp = dfs(verticesAdya, color, w, v, ...);
9.        Procesamiento de retroceso de la arista de árbol vw, empleando wResp (como en
          orden interno)
10.    else if (color[w] == gris && w ≠ p)
11.       Verificación (o sea, procesamiento) de la arista de retorno vw
       // si no, wv ya se recorrió, así que se hace caso omiso de vw.
12.    adyaRest = resto(adyaRest)
13. Procesamiento en orden posterior del vértice v, incluido el cálculo final de respuesta
14. color[v] = negro;
15. return respuesta;
```

Análisis

El tiempo de ejecución y las necesidades de espacio son los mismos del algoritmo 7.3: el tiempo está en $\Theta(n + m)$ y el espacio extra para el arreglo `color` está en $\Theta(n)$. La aplicación podría agregar código que eleve el orden asintótico, pero si todos los enunciados insertados se ejecutan en tiempo constante, el tiempo seguirá siendo lineal.

Búsqueda primero en amplitud no dirigida

Al igual que en la búsqueda primero en profundidad, en una búsqueda primero en amplitud en un grafo no dirigido surge la cuestión de reprocesar una arista no dirigida. Una solución sencilla sería tratar el grafo no dirigido como grafo dirigido simétrico. No conocemos aplicaciones de la búsqueda primero en amplitud en las que tal tratamiento sería incorrecto. Cada una de las aristas se procesa una vez en la dirección "hacia adelante", así que en el caso de una arista no dirigida la dirección en la que se le encuentra primero se considera "hacia adelante" durante la búsqueda. Cuando se encuentre la arista en la otra dirección, conducirá a un vértice ya descubierto, y normalmente se hará caso omiso de ella. No obstante, véase el ejercicio 7.7.

7.7 Componentes biconectados de un grafo no dirigido

En la sección 7.2 planteamos estas preguntas:

1. Si el aeropuerto de una ciudad está cerrado por mal tiempo, ¿se puede volar todavía entre todos los demás pares de ciudades?

2. Si una computadora de una red se "cae", ¿sigue siendo posible enviar mensajes entre todos los demás pares de computadoras de la red?

En esta sección sólo consideraremos grafos no dirigidos. Planteada como problema de grafos, la pregunta es:

Problema 7.1

Si se elimina cualquier vértice (y las aristas que inciden en él) de un grafo conectado, ¿el subgrafo residual sigue estando conectado? ■

Esta pregunta es importante en grafos que representan todo tipo de redes de comunicación o de transporte. También es importante hallar los vértices, en su caso, cuya eliminación pueda desconectar el grafo. El propósito de esta sección es presentar un algoritmo eficiente para contestar estas preguntas. El algoritmo fue descubierto por R. E. Tarjan y fue uno de los primeros en demostrar la gran potencia de la búsqueda primero en profundidad.

7.7.1 Puntos de articulación y componentes biconectados

Primero estableceremos algo de terminología y propiedades básicas.

Definición 7.21 Componente biconectado

Decimos que un grafo no dirigido conectado G está *biconectado* si sigue estando conectado después de eliminarse uno cualquiera de sus vértices y las aristas que inciden en ese vértice.

Un *componente biconectado* (o *bicomponente*, para abreviar) de un grafo no dirigido es un subgrafo biconectado máximo, es decir, un subgrafo biconectado que no está contenido en ningún subgrafo biconectado de mayor tamaño. ■

Definición 7.22 Punto de articulación

Un vértice v es un *punto de articulación* (también llamado *punto de corte*) de un grafo no dirigido G si existen dos vértices distintos w y x (distintos también de v) tales que v esté en todos los caminos que van de w a x. ■

Es evidente que la eliminación de un punto de articulación deja un grafo no conectado, así que un grafo conectado está *biconectado* si y sólo si no tiene puntos de articulación. La figura 7.22 ilustra los componentes biconectados. Obsérvese que, aunque los componentes biconectados dividen las aristas en conjuntos disjuntos, no dividen los vértices; algunos vértices están en más de un componente. (¿Cuáles son?)

Hay una caracterización alterna de los componentes biconectados, en términos de una relación de equivalencia de las aristas, que en ocasiones es útil. Dos aristas e_1 y e_2 son equivalentes si $e_1 = e_2$ o si existe un ciclo simple que contiene tanto a e_1 como a e_2. Entonces, todo subgrafo que consista en las aristas de una clase de equivalencia y los vértices incidentes será un componente biconectado. (Se deja como ejercicio verificar que la relación descrita es en verdad una relación de equivalencia y verificar que caracteriza a los componentes biconectados; véase el ejercicio 7.34.)

Las aplicaciones que nos motivan para estudiar la biconectividad deben sugerir un problema doble: cómo determinar si existe una *arista* cuya eliminación desconectaría un grafo, y cómo en-

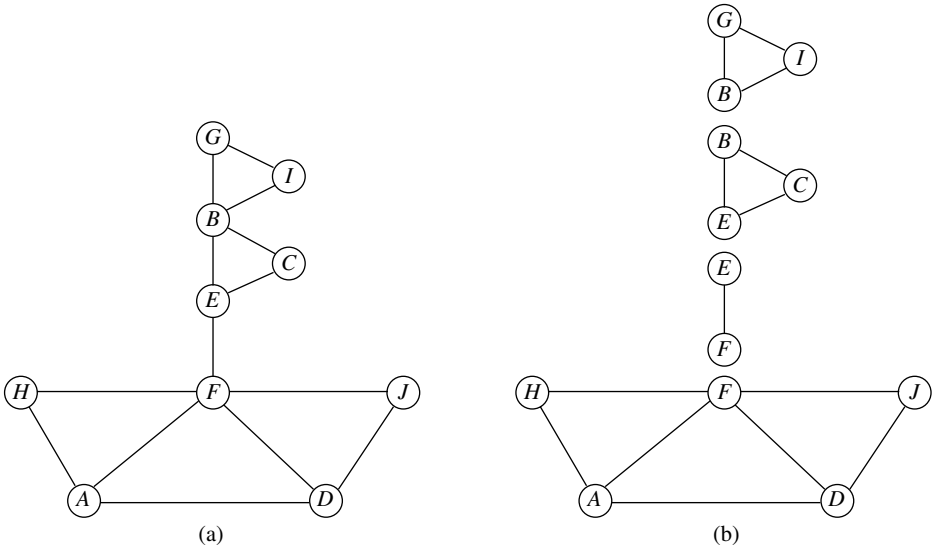


Figura 7.22 (a) Grafo no dirigido. (b) Sus componentes biconectados

contrar tal arista si existe. Por ejemplo, si una vía de ferrocarril se daña, ¿los trenes todavía pueden viajar entre cualesquier dos estaciones? Las relaciones entre los dos problemas se examinan en el ejercicio 7.41.

El algoritmo que usaremos para hallar componentes biconectados emplea el esqueleto de búsqueda primero en profundidad del algoritmo 7.8 y la idea de un árbol de búsqueda primero en profundidad de la sección 7.4.3. Durante la búsqueda, se calculará y almacenará información que permita dividir las aristas (e, implícitamente, los vértices incidentes) en componentes biconectados conforme avanza la búsqueda. ¿Qué información hay que guardar? ¿Cómo se usa para determinar los componentes biconectados? Existen varias respuestas *erróneas* a estas preguntas que parecen razonables si no se les examina minuciosamente. Dos aristas están en el mismo componente biconectado si están en un ciclo simple, y todo ciclo debe incluir por lo menos una arista de retorno. Recomendamos al lector resolver el ejercicio 7.35 antes de continuar; es necesario examinar varios ejemplos para determinar relaciones entre las aristas de retorno y los componentes biconectados.

De aquí en adelante usaremos el término más corto *bicomponente* en lugar de *componente biconectado*.

7.7.2 El algoritmo de bicomponentes

El procesamiento de los vértices durante una búsqueda primero en profundidad puede efectuarse cuando se *descubre* un vértice (en orden previo, línea 2 del esqueleto del algoritmo 7.8), cuando la búsqueda retrocede *a* él (en orden interno, línea 9 del esqueleto) y justo antes de terminársele (en orden posterior, línea 13 del esqueleto). El algoritmo de bicomponentes prueba si un vértice del árbol de búsqueda primero en profundidad es un punto de articulación cada vez que la búsqueda retrocede *a* él. Todas las referencias a árboles en esta explicación son al árbol de búsqueda

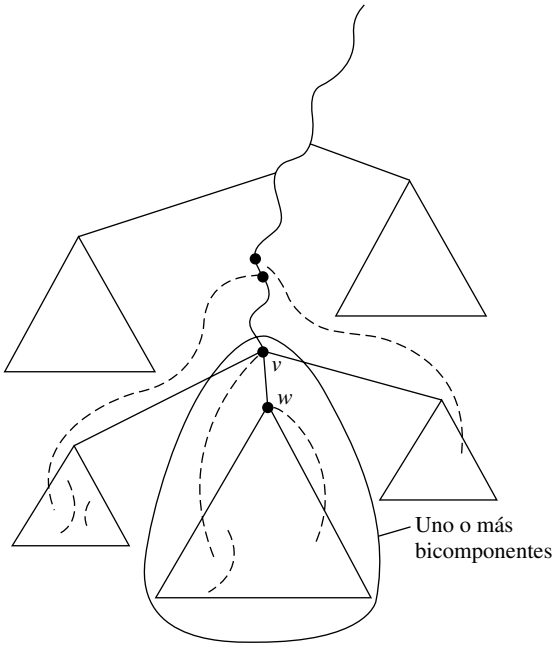


Figura 7.23 Punto de articulación en un árbol de búsqueda primero en profundidad

primero en profundidad. Recordemos que, en una búsqueda primero en profundidad de un grafo no dirigido, todas las aristas son de árbol o bien de retorno.

Supóngase que la búsqueda está retrocediendo a v de w. Si no existe una arista de retorno que vaya de cualquier vértice del subárbol cuya raíz es w a un antepasado propio de v, ello implica que v está en todos los caminos de G que van de la raíz del árbol DFS a w y por tanto es un punto de articulación. En la figura 7.23 se presenta una ilustración. (Cabe señalar que este argumento no es válido si v es la raíz.) El subárbol cuya raíz es w, junto con todas las aristas de retorno que salen de él y también la arista vw, se pueden separar del resto del grafo en v, pero no es necesariamente un bicomponente; podría ser una unión de varios bicomponentes. Garantizamos la separación correcta de los bicomponentes quitando cada uno tan pronto como lo detectamos. Los vértices que están en los extremos del árbol se prueban antes que los vértices más cercanos a la raíz, para determinar si son o no puntos de articulación. Ello garantiza que, al hallarse un punto de articulación, el subárbol en cuestión (junto con las aristas adicionales arriba mencionadas) forma un bicomponente.

Esta explicación sugiere que el algoritmo debe saber qué tanto es posible retroceder en el árbol desde cada vértice siguiendo aristas de árbol (que implícitamente se alejan de la raíz) y ciertas aristas de retorno. Esta información se almacenará en una variable local, `retro`. (Hay una copia distinta de `retro` para cada vértice activo.) Cuando un vértice termina su búsqueda, devuelve su valor final de `retro` al invocador. El procedimiento de búsqueda primero en profundidad calculará `tiempoDescubrir` y `tiempoTerminar` tal como se describe en la definición 7.15. Los valores de `retro` serán estos `tiempoDescubrir`. Para un vértice v, se podría asignar un valor a `retro` (o se le podría modificar) cuando:

1. se descubre y visita v (orden previo), para inicializar `retro`;
2. la búsqueda está tratando de explorar, pero se topa con una arista de retorno que viene de v (como en la figura 7.24(b) con $v = F$, en la figura 7.24(c) con $v = C$, y en la figura 7.24(e) con $v = I$);
3. la búsqueda retrocede a v (como en la figura 7.24(d) con $v = B$ y en la figura 7.24(f) con $v = G$), ya que es posible llegar desde v a cualquier vértice al que se puede llegar desde un hijo de v.

Es fácil determinar cuál de dos vértices está más atrás en el árbol: si v es un antepasado propio de w, entonces `tiempoDescubrir[v] < tiempoDescubrir[w]`. Así pues, podemos formular las reglas siguientes para establecer el valor de `retro`:

1. En orden previo, `retro = tiempoDescubrir[v]` (pero véase el ejercicio 7.38).
2. Al tratar de explorar desde v y detectarse una arista de retorno vw, `retro = min(retro, tiempoDescubrir[w])`.
3. Al retroceder de w a v, digamos que el valor devuelto por la visita a w es `wRetro`. Entonces, para v, `retro = min(retro, wRetro)`.

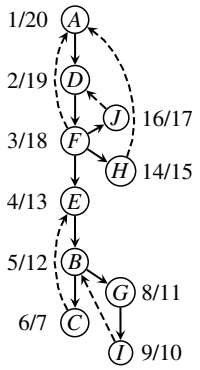
La condición que se prueba para detectar un bicomponente al retroceder de w a v es
`wRetro` $\geq$ `tiempoDescubrir[v]`.
(Esta condición se prueba pero no se satisface en las figuras 7.24(d) y 7.24(f); se satisface en las figuras 7.24(g) y 7.24(h).) Si la prueba se satisface, quiere decir que v es un punto de articulación (excepto, quizá, si v es la raíz del árbol); se ha hallado un posible bicomponente y podríamos dejarlo de considerar.

El problema de exactamente cuándo y cómo probar si hay bicomponentes es sutil pero crucial para que un algoritmo sea correcto. (Véase el ejercicio 7.40.) La esencia del argumento de corrección está contenida en el teorema siguiente.

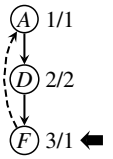
Teorema 7.13 En un árbol de búsqueda primero en profundidad, un vértice v que no es la raíz es un punto de articulación si y sólo si v no es una hoja y en algún subárbol de v no incide ninguna arista de retorno proveniente de un antepasado propio de v.

Demostración (Sólo si) Supóngase que v, un vértice distinto de la raíz, es un punto de articulación. Entonces existen vértices x y y tales que v, x y y son distintos y v está en todos los caminos que van de x a y. Al menos uno de los dos, x o y, debe ser un descendiente propio de v, pues de otro modo habría un camino entre ellos que pasaría por aristas (no dirigidas) del árbol que no pasan por v. Por tanto, v no es una hoja. Supóngase ahora que todos los subárboles de v tienen una arista de retorno que va a un antepasado propio de v; nuestra afirmación es que ello contradice el supuesto de que v es un punto de articulación. Hay dos casos: cuando sólo x o sólo y (pero no los dos) es un descendiente de v, y cuando ambos son descendientes de v. Para el primer caso, los caminos entre x y y que no pasan por v se ilustran en la figura 7.25. Dejamos el otro caso como ejercicio. La parte "si" de la demostración también se deja como ejercicio. □

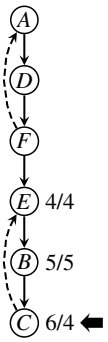
El teorema 7.13 no nos dice en qué condiciones la raíz es un punto de articulación. Véase el ejercicio 7.37.



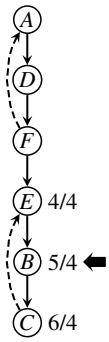
(a) El árbol de búsqueda primero en profundidad completo (líneas continuas) y las aristas de retorno (líneas punteadas).



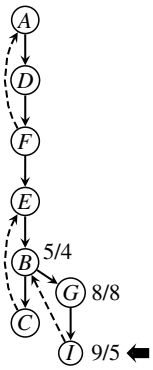
(b) Ir hacia adelante; inicializar valores de `retro`. Detectar arista de retorno *FA*; actualizar retro en *F*.



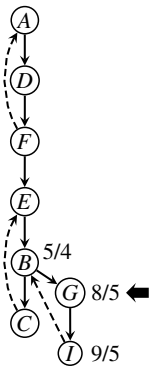
(c) Continuar hacia adelante. Detectar arista de retorno *CE*; actualizar `retro` en *C*.



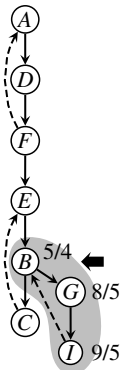
(d) Devolver 4 de *C* a *B*, que es menor que `tiempoDescubrir[B]`. Actualizar `retro` en *B*.



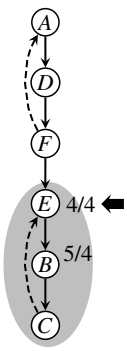
(e) Hacia adelante a *G* e *I*; detectar arista de retorno *IB*; actualizar `retro` en *I*.



(f) Devolver 5 de *I* a *G*, que es menor que `tiempoDescubrir[G]`. Actualizar retro en *G*.



(g) Devolver 5 de *G* a *B*, que es igual que `tiempoDescubrir[B]`. Quitar bicomponente.



(h) Devolver 4 de *B* a *E*, que es igual que `tiempoDescubrir[E]`. Quitar bicomponente.

Figura 7.24 Acción del algoritmo de bicomponentes sobre el grafo de la figura 7.22 (detección de los dos primeros bicomponentes): la parte (a) muestra los tiempos de descubrimiento y de terminación. Los rótulos de los vértices en las partes (b) a (h) son `tiempoDescubrir/retro`.

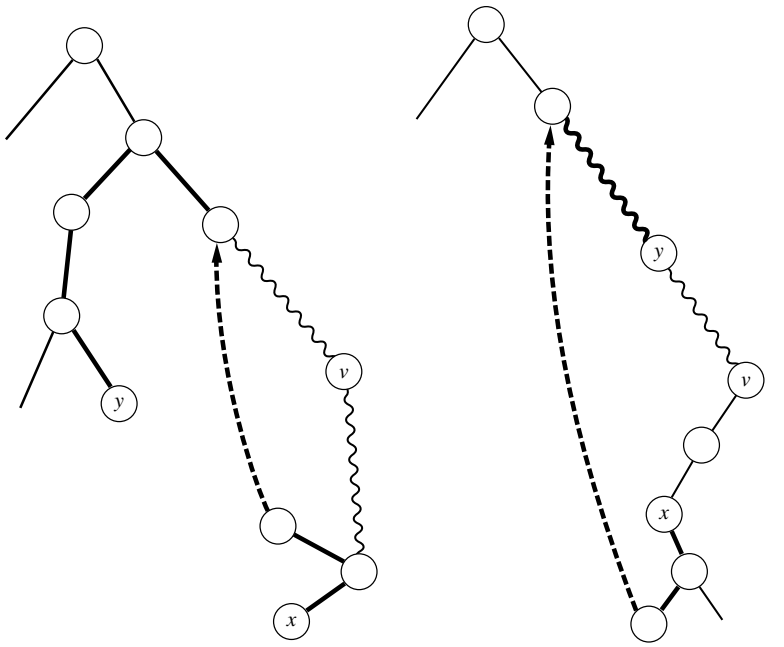


Figura 7.25 Ejemplos de la demostración del teorema 7.13. Las líneas onduladas denotan caminos

Ahora podemos delinear el trabajo que debe efectuarse en la búsqueda primero en profundidad. Cabe señalar que se insertará código en el esqueleto de búsqueda primero en profundidad no dirigida del algoritmo 7.8. No obstante, antes de desarrollar el algoritmo completo, conviene bosquejarlo en un nivel alto.

```
int bicompDFS(v) // BOSQUEJO
    color[v] = gris;
    tiempo ++; tiempoDescubrir[v] = tiempo;
    retro = tiempoDescubrir[v];
    while (existe una arista no recorrida vw)
       Si vw es una arista de árbol:
          wRetro = bicompDFS(w);
          // Ahora retrocedemos a v
          if (wRetro >= tiempoDescubrir[v])
             Enviar a la salida un nuevo bicomponente que consiste en el subárbol cuya
             raíz es w y las aristas incidentes, pero sin incluir las aristas de bicomponentes
             que se enviaron a la salida antes.
       retro = min(retro, wRetro);
    si no, si vw es una arista de retorno:
       retro = min(tiempoDescubrir[w], retro);
       // Continuar el ciclo while.
return retro;
```

El algoritmo debe recordar qué aristas recorrió durante la búsqueda para poder identificar fácilmente las que pertenecen a un bicomponente y excluirlas de consideración en el momento apropiado. Como ilustra el ejemplo de la figura 7.24, cuando se detecta un bicomponente, sus aristas son las que se procesaron más recientemente. Por ello, las aristas se apilan en `pilaAristas` conforme se van encontrando. Cuando se detecta un bicomponente al retroceder de, digamos, w a v, las aristas de ese bicomponente son las que están en la pila entre el tope y vw (inclusive). En ese momento ya pueden desapilarse esas aristas.

La incorporación del bosquejo al esqueleto del algoritmo 7.8, junto con cierto código de control de nivel superior, produce el algoritmo final. (Se calcula `tiempoTerminar` por consistencia con Rastreo DFS, que es el algoritmo 7.4, pero podría omitirse.)

Algoritmo 7.9 Componentes biconectados

Entradas: Un arreglo `verticesAdya` de listas de adyacencia para un grafo no dirigido $G = (V, E)$; n, el número de vértices. También se usan los arreglos globales `tiempoDescubrir` y `tiempoTerminar`, y una variable global `tiempo`. Todos los arreglos deben estar definidos para los índices $1, \ldots, n$; el elemento número 0 no se usa.

Salidas: Conjuntos (por ejemplo, listas) de las aristas que pertenecen a cada componente biconectado de G.

Comentarios: Se usan las operaciones del TDA Pila descritas en la sección 2.4.1. Los significados de los colores son: blanco = no descubierto, gris = activo, negro = terminado.

Procedimiento: Véase la figura 7.26. ■

Puesto que `pilaAristas` podría crecer hasta el número de aristas de G, se sugiere una implementación flexible, tal vez basada en el TDA Lista.

7.7.3 Análisis

Como es costumbre, $n = |V|$ y $m = |E|$. La inicialización que se efectúa en `bicomponentes` incluye $\Theta(n)$ operaciones. `bicompDFS` es el esqueleto de búsqueda primero en profundidad no dirigida al que se ha añadido el procesamiento apropiado de vértices y aristas. El esqueleto de búsqueda primero en profundidad no dirigida tarda un tiempo en $\Theta(n + m)$. El espacio ocupado está en $\Theta(n + m)$.

Por tanto, si la cantidad de procesamiento que se efectúa con cada vértice y arista está acotada por una constante, la complejidad de `bicomponentes` estará en $\Theta(n + m)$. Es fácil ver que tal es el caso. El único punto en el que la observación necesaria no es trivial es cuando la búsqueda retrocede de w a v. A veces se ejecuta el ciclo de *salida* que desapila aristas de `pilaAristas` y a veces no, y el número de aristas que se desapilan en cada ocasión varía. No obstante, cada arista se apila y se desapila exactamente una vez, así que, en general, la cantidad de trabajo efectuada está en $\Theta(m)$.

7.7.4 Generalizaciones

El prefijo *bi* significa "dos". En términos informales, un grafo biconectado tiene dos caminos de vértices disjuntos entre cualquier par de vértices (véase el ejercicio 7.33). Podemos definir la triconectividad (y, en general, la k-conectividad) como la propiedad de tener tres (o k, en general) caminos de vértices disjuntos entre cualquier par de vértices. Se ha desarrollado un algoritmo eficiente que usa búsqueda primero en profundidad para hallar los componentes triconectados de un

grafo (véanse las Notas y referencias al final del capítulo), pero es mucho más complicado que el algoritmo para bicomponentes.

```
void bicomponentes(ListaInt[] verticesAdya, int n)
   int v;
   PilaInt pilaAristas;
   int[] color = new int[n+1];

   Inicializar el arreglo color con blanco para todos los vértices.
   tiempo = 0;
   pilaAristas = crear();
   for (v = 1; v <= n; v ++)
     if (color[v] == blanco)
       bicompDFS(verticesAdya, color, v, −1);
   return;

int bicompDFS(ListaInt[] verticesAdya, int[] color, int v, p)
   int w;
   ListaInt adyaRest;
   int retro;

 1. color[v] = gris;
2a. tiempo ++; tiempoDescubrir[v] = tiempo;
2b. retro = tiempoDescubrir[v];
 3. adyaRest = verticesAdya[v];
 4. while (adyaRest <> nil)
 5.   w = primero(adyaRest);
 6.   if (color[w] == blanco)
 7.     push(pilaAristas, vw)
 8.     int wRetro = bicompDFS(verticesAdya, color, w, v);
9a.     // Procesamiento en retroceso de la arista de árbol vw
9b.     if (wRetro >= tiempoDescubrir[v])
9c.       Inicializar para nuevo bicomponente.
9d.       Desapilar y enviar a la salida pilaAristas desde el tope hasta vw.
9e.     retro = min(retro, wRetro);
 10.  else if (color[w] == gris && w <> p)
11a.     // Procesar arista de retroceso vw.
11b.     push(pilaAristas, vw);
11c.     retro = min(tiempoDescubrir[w], retro);
       // de lo contrario ya se recorrió wv, así que se pasa por alto vw.
12.    adyaRest = resto(adyaRest);
13. tiempo ++; tiempoTerminar[v] = tiempo;
14. color[v] = negro;
15. return retro;
```

Figura 7.26 Procedimiento para el algoritmo 7.9

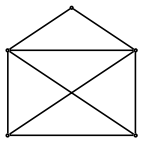
Ejercicios

Sección 7.2 Definiciones y representaciones

7.1 Dibuje un grafo unidirigido conectado (cuyas aristas podrían representar calles de dos sentidos) tal que cada vértice esté en algún ciclo no dirigido, pero que independientemente de la orientación que se dé a las aristas (es decir, que se conviertan en aristas dirigidas, o calles de un solo sentido) el grafo no esté fuertemente conectado.

7.2 Este ejercicio trata los caminos de Euler.

a. Un juego muy popular entre los niños de primaria consiste en dibujar la figura que sigue sin despegar el lápiz del papel y sin volver a pasar por ninguna línea. Inténtelo.



b. La figura 7.27 presenta un problema similar pero un poco más difícil: muestra un río con dos islas conectadas entre sí y con las riberas mediante siete puentes. El problema consiste en determinar si hay alguna manera de dar un paseo que parta de cualquier orilla del río o de cualquier isla y cruce cada uno de los puentes exactamente una vez. (No se permite nadar.) Inténtelo.

*** c.** Los problemas de las partes (a) y (b) se pueden estudiar en abstracto examinando los grafos siguientes. G_2 se obtiene representando cada ribera e isla como un vértice y cada puente como una arista. (Algunos pares de vértices están conectados por dos aristas, pero esta divergencia respecto a la definición de grafo no causa problemas aquí.) El problema general es: dado un grafo (en el que se permiten múltiples aristas entre pares de vértices), hallar un camino por el grafo que recorra cada arista exactamente una vez. Semejante camino se denomina camino de Euler. Este problema se puede resolver para G_1 pero no para G_2. Es decir, no hay

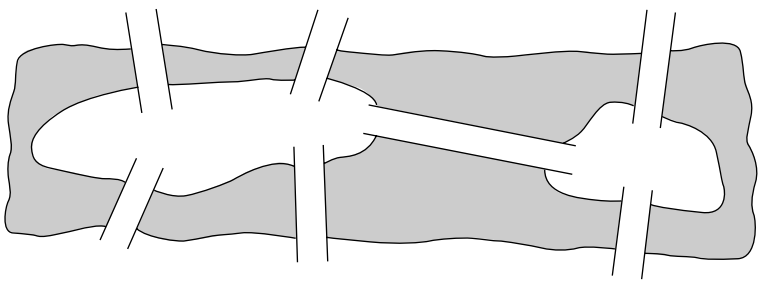
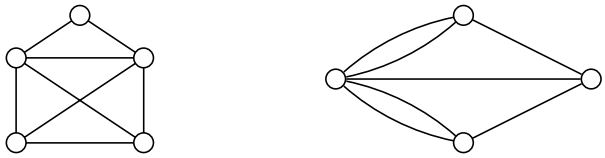


Figura 7.27 Los puentes de Königsberg

manera de caminar por cada uno de los puentes exactamente una vez. Encuentre una condición necesaria y suficiente para que un grafo tenga un camino de Euler.



G_1 G_2

7.3 Supóngase que un grafo dirigido G representa una relación binaria R. Describa una condición en términos de G que se cumpla si y sólo si R es transitiva.

Sección 7.3 Recorrido de grafos

7.4 Determine el árbol de búsqueda primero en profundidad para el grafo empleado en el ejemplo 7.6 (vea la figura 7.28) con G como vértice de partida y haciendo uno de dos supuestos acerca del orden dentro de las listas de adyacencia:

a. Cada lista de adyacencia está en orden alfabético.
b. Cada lista de adyacencia está en orden alfabético inverso.

7.5 Determine el árbol de búsqueda primero en amplitud y las distancias primero en amplitud para el grafo empleado en el ejemplo 7.7 (véase la figura 7.28) con G como vértice de partida y haciendo uno de dos supuestos acerca del orden dentro de las listas de adyacencia:

a. Cada lista de adyacencia está en orden alfabético.
b. Cada lista de adyacencia está en orden alfabético inverso.

7.6 Sea G un grafo conectado, y sea s un vértice de G. Sea T_D un árbol de búsqueda primero en profundidad que se forma efectuando una búsqueda primero en profundidad en G partiendo de s. Sea T_B un árbol abarcante primero en amplitud que se forma efectuando una búsqueda primero en amplitud en G partiendo de s. ¿Siempre se cumple que altura(T_D) $\geq$ altura(T_B)? ¿Importa si el grafo es dirigido o no dirigido? Presente un argumento claro o un contraejemplo.

7.7 Demuestre que cuando se efectúa una búsqueda primero en amplitud en un grafo no dirigido todas las aristas del grafo son una arista de árbol o bien una arista cruzada. (Una arista cruza-

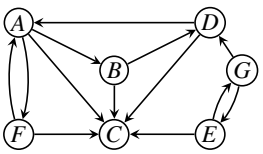


Figura 7.28 Grafo dirigido de los ejemplos 7.6 y 7.7, empleado en varios ejercicios

da en una búsqueda primero en amplitud es una arista entre dos vértices tales que ninguno es descendiente del otro en el árbol abarcante primero en amplitud.)

Sección 7.4 Búsqueda de primero en profundidad en grafos dirigidos

7.8 Bosqueje un algoritmo para calcular el grafo transpuesto, dado el grafo original en forma de un arreglo de listas de adyacencia. Su algoritmo se deberá ejecutar en tiempo lineal.

a. Escriba pseudocódigo para el procedimiento y cualesquier subrutinas.

b. Muestre cómo opera su algoritmo con la figura 7.28, suponiendo que las listas de adyacencia del grafo original están en orden alfabético. Especifique el orden de los vértices en las listas de adyacencia del grafo transpuesto. (Recuerde, no los va a ordenar; ello podría ser costoso.)

7.9 Clasifique las aristas del grafo empleado en el ejemplo 7.6 (vea la figura 7.28) según la definición 7.14, suponiendo que la búsqueda primero en profundidad se inicia en el vértice *G*, y que los vértices adyacentes se procesan en orden alfabético.

7.10 En el caso 2 de la definición 7.14 (arista de retorno), ¿qué color(es) puede tener w cuando se verifica la arista vw?

7.11 Ejecute Rastreo DFS (algoritmo 7.4) con el grafo dirigido de la figura 7.29, y clasifique todas las aristas.

a. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

b. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

c. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

d. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

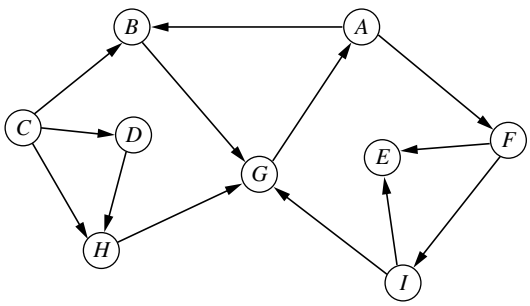


Figura 7.29 Grafo dirigido para los ejercicios 7.11 y 7.23

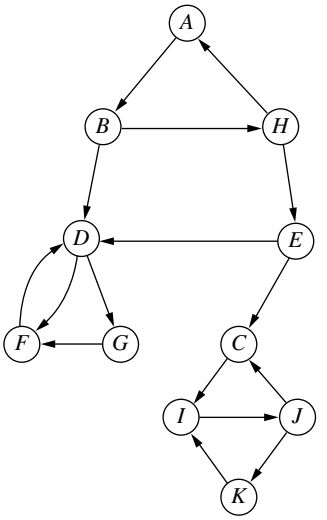


Figura 7.30 Grafo dirigido para los ejercicios 7.12 y 7.24

7.12 Ejecute Rastreo DFS (algoritmo 7.4) con el grafo dirigido de la figura 7.30, y clasifique todas las aristas.

a. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

b. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

c. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

d. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

7.13 Dé un ejemplo de grafo en el que una búsqueda primero en profundidad retroceda de un vértice antes de que se descubran todos los vértices a los que se puede llegar desde él por una o más aristas.

7.14 Suponga que v y w son vértices distintos del mismo árbol dirigido, pero no tienen relación antepasado/descendiente. Demuestre que existe un tercer vértice c, su mínimo común antepasado, tal que existen caminos en el árbol que van de c a v y de c a w, y que dichos caminos no tienen aristas en común. *Sugerencia:* Utilice el hecho de que cada vértice de un árbol tiene exactamente un camino que va de la raíz a él.

7.15 Demuestre el inciso 3 del teorema 7.1.

7.16 Describa cómo modificaría el esqueleto DFS para obtener un algoritmo para un grafo dirigido cuya salida es una lista de las aristas del árbol de búsqueda primero en profundidad.

7.17

a. Escriba un algoritmo para determinar si un grafo dirigido tiene un ciclo.

b. Si usó búsqueda primero en profundidad en el algoritmo anterior, trate de escribir un algoritmo para el mismo problema empleando búsqueda primero en amplitud, y viceversa. ¿Considera que hay razones de peso para preferir cualquiera de las estrategias de búsqueda con este problema?

7.18 Muestre el resultado del algoritmo 7.4 e indique qué números topológicos asigna el algoritmo 7.5 si el grafo de dependencia definido en el ejemplo 7.14 se procesa en orden inverso. Es decir, suponga que el ciclo `for` de `barridoDfs` va de 9 a 1, y que las listas de adyacencia también están en orden inverso.

7.19 Para cada grafo de la figura 7.18, ejecute manualmente el algoritmo 7.5 con la modificación que lo hace calcular un orden topológico en lugar de un orden topológico inverso. Suponga que los vértices están en orden numérico en las listas de adyacencia. También verifique durante la ejecución si el grafo tiene un ciclo (¿qué condición deberá verificar durante `dfs`?). Deténgase tan pronto como se detecte un ciclo y explique cómo se detectó, o determine el orden topológico completo si no hay ciclos. Compare su orden topológico con el orden topológico inverso del ejemplo 7.15 (que usó el grafo transpuesto). ¿Son iguales?

7.20 Decimos que un DAG es una *retícula* si existe un vértice desde el que se pueda llegar a todos los vértices y un vértice al que se pueda llegar desde todos los vértices.

a. Bosqueje un algoritmo para determinar si un DAG es una retícula.

b. ¿Qué orden asintótico tiene su tiempo de ejecución?

c. Muestre el funcionamiento de su algoritmo con el grafo del ejemplo 7.15. ¿Es una retícula ese grafo?

⋆ **7.21** Otra estrategia para el ordenamiento topológico consiste en recordar los vértices "origen". En un principio, cada vértice tiene un *grado de entrada* que es el número de aristas dirigidas que *entran* en el vértice. Un *origen* es un vértice con grado de entrada 0. De lo que se trata es de asignar números topológicos en orden ascendente a los vértices origen. Cada vez que se numera un vértice v, se debe reducir el grado de entrada de todos los vértices en los que entra una arista procedente de v. Es como si v se sacara del grafo después de numerarse. A medida que otros grados de entrada se reducen a 0, otros vértices se convierten en orígenes. Escriba un algoritmo para implementar esta estrategia. Especifique las estructuras de datos que necesita para llevar la contabilidad. ¿Qué orden asintótico tiene su algoritmo con un DAG de n vértices y m aristas?

Sección 7.5 Componentes fuertemente conectados de un grafo dirigido

7.22 Demuestre que la condensación de un grafo dirigido es acíclica.

7.23 Halle los componentes fuertes del grafo dirigido de la figura 7.29 siguiendo cuidadosamente los pasos del algoritmo. (Resulta útil calcular `tiempoDescubrir` y `tiempoTerminar` de los vértices, aunque el algoritmo no lo exige.)

a. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

b. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

c. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

d. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

7.24 Halle los componentes fuertes del grafo dirigido de la figura 7.30 siguiendo cuidadosamente los pasos del algoritmo. (Resulta útil calcular `tiempoDescubrir` y `tiempoTerminar` de los vértices, aunque el algoritmo no lo exige.)

a. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

b. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético.

c. Suponga que los vértices están indizados en orden alfabético en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

d. Suponga que los vértices están indizados en orden alfabético inverso en el arreglo `verticesAdya` y que todas las listas de adyacencia están en orden alfabético inverso.

7.25 Extienda o modifique el algoritmo de componentes fuertes para que produzca una lista de todas las aristas y todos los vértices de cada componente fuerte. Trate de reducir al mínimo el tiempo extra que se requiere para hacerlo.

7.26 ¿Cualquiera de las búsquedas primero en profundidad del algoritmo de componentes fuertes se puede sustituir (fácilmente) por una búsqueda primero en amplitud? Explique por qué sí o por qué no.

Sección 7.6 Búsqueda de primero en profundidad en grafos no dirigidos

7.27 Escriba un algoritmo de búsqueda primero en profundidad para un grafo no dirigido tal que la salida sea una lista de las aristas encontradas, en la que cada arista aparezca una sola vez.

7.28 Demuestre que si G es un grafo no dirigido conectado, cada una de sus aristas está en el árbol de búsqueda primero en profundidad o bien es una arista de retorno.

7.29

a. Escriba un algoritmo para determinar si un grafo no dirigido tiene un ciclo.

b. Si usó búsqueda primero en profundidad en el algoritmo anterior, trate de escribir un algoritmo para el mismo problema empleando búsqueda primero en amplitud, y viceversa. ¿Considera que hay razones de peso para preferir cualquiera de las estrategias de búsqueda con este problema?

c. ¿En qué difieren estos algoritmos, si acaso, de los del ejercicio 7.17?

7.30 Describa un algoritmo para determinar si un grafo no dirigido $G = (V, E)$, con $n = |V|$ y $m = |E|$, es un árbol. ¿Usaría el mismo algoritmo si pudiera suponer que el grafo está conectado? Si no, describa uno que haga también ese supuesto.

⋆ **7.31** Considere el problema de hallar la longitud del ciclo más corto de un grafo no dirigido. A continuación proponemos una solución que no es correcta. Explique por qué no siempre funciona.

> Cuando se encuentra una arista de retorno, digamos vw, durante una búsqueda primero en profundidad, forma un ciclo con las aristas de árbol que van de w a v. La longitud del ciclo es `profundidad[v] − profundidad[w] + 1`, donde `profundidad` es la profundidad en el árbol DFS. Así pues, efectúe una búsqueda primero en profundidad, recordando la profundidad de cada vértice. Cada vez que encuentre una arista de retorno, calcule la longitud del ciclo y guárdela si es menor que la longitud más corta hallada previamente.

Busque un defecto fundamental en la estrategia, no un detalle.

Sección 7.7 Componentes biconectados de un grafo no dirigido

7.32 Enumere los puntos de articulación del grafo cuyo árbol de búsqueda primero en profundidad se muestra en la figura 7.31.

7.33 La siguiente propiedad de un grafo $G = (V, E)$ ¿es necesaria y suficiente para que G esté biconectado? Demuestre su respuesta.

> Por cada par de vértices distintos v y w en V, hay dos caminos distintos de v a w que no tienen vértices en común salvo v y w.

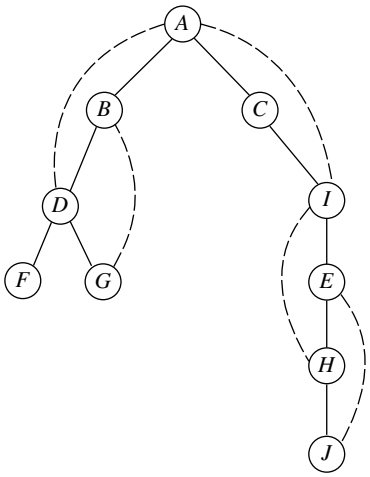
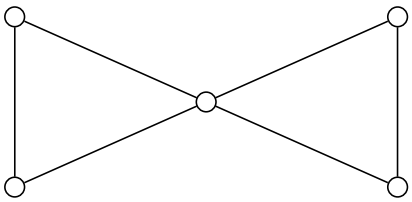


Figura 7.31 Árbol de búsqueda primero en profundidad para el ejercicio 7.32

7.34 Para un grafo no dirigido $G = (V, E)$, considere la relación siguiente, R, entre las aristas de E: $e_1 R e_2$ si y sólo si $e_1 = e_2$ o existe un ciclo simple en G que contiene a e_1 y e_2.

a. Demuestre que R es una relación de equivalencia.

b. ¿Cuántas clases de equivalencia hay en este grafo?



c. Demuestre que un subgrafo que consiste en las aristas que están en una clase de equivalencia de la relación R y los vértices incidentes es un subgrafo biconectado máximo de G.

★ **7.35** Las dos definiciones que siguen, de funciones de los vértices de un árbol de búsqueda primero en profundidad de un grafo no dirigido son intentos de establecer condiciones necesarias y/o suficientes para que dos vértices estén en el mismo componente biconectado del grafo. Demuestre mediante presentación de contraejemplos que dichos intentos fracasan.

a. Definimos $viejo_1(x)$ = el antepasado "más viejo" —es decir, más cercano a la raíz— de x al que se puede llegar siguiendo aristas de árbol (alejándose de la raíz) y aristas de retorno; o $viejo_1(x) = x$ si ningún camino de ese tipo conduce a un antepasado de x. Demuestre que $viejo_1(v) = viejo_1(w)$ no es necesario ni suficiente para que v y w estén en el mismo bicomponente.

b. Definimos $viejo_2(x)$ = el antepasado más viejo de x al que se puede llegar siguiendo aristas de árbol dirigidas (alejándose de la raíz) y *una* arista de retorno; o $viejo_2(x) = x$ si ningún camino de ese tipo conduce a un antepasado de x. Demuestre que $viejo_2(v) = viejo_2(w)$ no es necesario ni suficiente para que v y w estén en el mismo bicomponente.

7.36 Complete la demostración del teorema 7.13.

7.37 Encuentre una condición necesaria y suficiente para que la raíz de un árbol de búsqueda primero en profundidad de un grafo conectado sea un punto de articulación. Demuéstrelo.

7.38 ¿El algoritmo de bicomponentes funcionaría correctamente si `retro` se inicializara con ∞ (o $2(n + 1)$) en lugar de `tiempoDescubrir[v]`? Explique su respuesta.

7.39 Dé un ejemplo de grafo que demuestre que el algoritmo de bicomponentes podría dar respuestas incorrectas si no se procura evitar que se apile una arista la segunda vez que se le encuentra en la estructura de listas de adyacencia. Esto equivale a tratar G como grafo dirigido simétrico en lugar de grafo no dirigido. La prueba que se efectúa en la línea 10 del algoritmo 7.9 se omitiría y esa línea sería un simple **`else`**.

7.40 ¿El algoritmo de bicomponentes funcionaría correctamente si la prueba para determinar bicomponentes se cambiara a `retro` $\geq$ `tiempoDescubrir[v]`? Si la respuesta es sí, explique por qué; si no, dé un ejemplo en el que no funcione.

7.41 Un grafo conectado está *biconectado por aristas* si no existe ninguna arista cuya eliminación desconecta el grafo. ¿Cuáles de las afirmaciones siguientes son verdad, si es que alguna lo es? Presente una demostración o un contraejemplo para cada una.

a. Un grafo biconectado está biconectado por aristas.

b. Un grafo biconectado por aristas está biconectado.

7.42 Suponga que G es un grafo conectado. Un *puente* es una arista de G cuya eliminación desconecta el grafo. Por ejemplo, en la figura 7.22 la arista EF es un puente. Dé un algoritmo para hallar los puentes en un grafo. ¿Qué complejidad de peor caso tiene su algoritmo?

Problemas adicionales

7.43 En la sección 7.2 mencionamos que, si un grafo se representa con una matriz de adyacencia, casi cualquier algoritmo que opere sobre el grafo tendrá una complejidad de peor caso en $\Omega(n^2)$, donde n es el número de vértices. No obstante, hay algunos problemas que se pueden resolver rápidamente, incluso si se usa la matriz de adyacencia. He aquí uno.

a. Sea $G = (V, E)$ un grafo dirigido con n vértices. Decimos que un vértice s es un *hipersumidero* si, por cada v en V tal que $s \neq v$, existe una arista vs y no hay aristas de la forma sv. Escriba un algoritmo para determinar si G tiene o no un hipersumidero, suponiendo que G está dado por su matriz de adyacencia $n \times n$.

b. ¿Cuántos elementos de matriz examina su algoritmo en el peor caso? Es fácil proponer un algoritmo que examina $\Theta(n^2)$ elementos, pero existe una solución lineal.

★ **7.44** Halle la mejor cota inferior que pueda para el número de elementos de matriz de adyacencia que es preciso examinar para resolver el problema descrito en el ejercicio 7.43. Demuestre que es una cota inferior. *Sugerencia:* Deberá ser fácil presentar un argumento claro para $2n - 2$. Se puede usar un argumento de adversario similar al de la sección 5.3.3 para obtener una cota inferior más categórica.

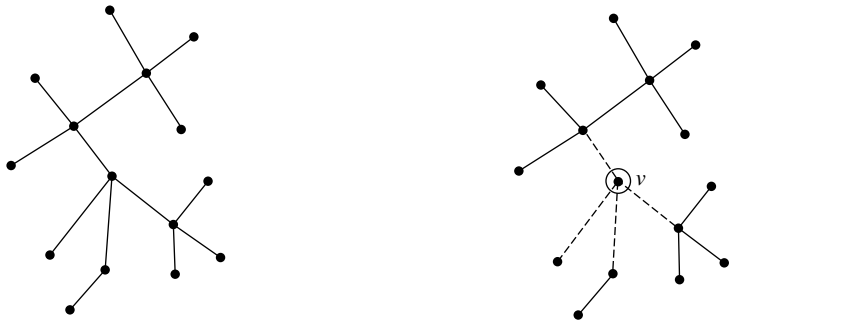
7.45 Diseñe un algoritmo eficiente para hallar un camino en un grafo no dirigido conectado que pase por cada arista exactamente una vez en cada dirección.

★ **7.46** Un *circuito de Euler* en un grafo no dirigido es un circuito (es decir, un ciclo que podría pasar por algunos vértices más de una vez) que incluye todas las aristas exactamente una vez. Escriba un algoritmo que halle un circuito de Euler en un grafo, o que indique que el grafo no contiene circuitos de Euler.

7.47 Considere la pregunta siguiente:

Problema 7.2

¿Existe un vértice v en G tal que se pueda llegar a cualquier otro vértice de G siguiendo un camino que parte de v? ■



(a) Un árbol, $n = 15$ (b) Subárboles que quedan cuando se elimina v

Figura 7.32 Ejemplo para el ejercicio 7.49

Si G es un grafo no dirigido, la pregunta se puede contestar fácilmente con una simple búsqueda primero en profundidad (o primero en amplitud) y una verificación para ver si se visitaron todos los vértices. Escriba un algoritmo para resolver el problema en el caso de un grafo dirigido. ¿Qué complejidad tiene su algoritmo?

7.48 Un grafo *bipartita* es un grafo cuyos vértices pueden dividirse en dos subconjuntos tales que no haya ninguna arista entre dos vértices cualesquiera del mismo subconjunto. Escriba un algoritmo para determinar si un grafo no dirigido es bipartita. ¿Qué complejidad de peor caso tiene su algoritmo?

★ **7.49** Cuando se eliminan de un árbol un vértice y las aristas que inciden en él, queda una colección de subárboles. Escriba un algoritmo que, al proporcionársele un grafo que es un árbol de n vértices, halle un vértice v cuya eliminación no deje ningún subárbol que tenga más de $n/2$ vértices. En la figura 7.32 se da un ejemplo. ¿Qué complejidad de peor caso tiene su algoritmo? (Deberá poder obtener una solución lineal.)

Programas

Cada uno de los programas siguientes que se pide al lector escribir requiere un procedimiento de *carga de grafo* que lee una descripción de un grafo de un archivo y prepara las listas de adyacencia. El apéndice A contiene un ejemplo de código en Java que puede servir como base. Suponga que las entradas contienen el número de vértices en la primera línea, seguido de una sucesión de líneas cada una de las cuales contiene un par de vértices que representan una arista. Escriba este procedimiento de modo que, con cambios pequeños, se pueda usar en cualquiera de los problemas.

Si desea una interfaz de usuario más elegante, haga que el grafo se cargue desde un archivo con nombre de modo que, una vez efectuada la carga, el usuario pueda introducir en la terminal "consultas" que piden al programa principal (*no* al procedimiento de carga de grafos mencionado) resolver un problema dado o producir una salida dada. En este caso, no olvide incluir una "consulta" para salir del programa.

Se recomienda escoger datos de prueba que ejerciten todos los aspectos del programa. Incluya algunos de los ejemplos del texto.

1. Un algoritmo de búsqueda primero en profundidad para determinar si un grafo no dirigido tiene un ciclo.
2. Un algoritmo de búsqueda primero en amplitud para determinar si un grafo dirigido tiene un ciclo.
3. El algoritmo de componentes fuertes descrito en el algoritmo 7.7.
4. El algoritmo de bicomponentes, algoritmo 7.9.

Notas y referencias

Tarjan sugirió la estructura de listas de adyacencia empleada en este capítulo, la cual se describe, junto con los algoritmos para componentes biconectados, ordenamiento topológico y muchos más, en Tarjan (1972) y Hopcroft y Tarjan (1973b). Hopcroft y Tarjan (1973a) presenta un algoritmo para hallar los componentes triconectados de un grafo. Hopcroft y Tarjan (1974) contiene un algoritmo muy eficiente para determinar si un grafo es plano: otro problema importante de grafos. El algoritmo de componentes fuertes (algoritmo 7.7) se debe a Sharir (1981). El uso de tres colores de vértices para "labores domésticas" en la búsqueda primero en profundidad, así como el uso de un solo contador de tiempo para `tiempoDescubrir` y `tiempoTerminar`, se deben a Cormen, Leiserson y Rivest (1990).

King y Smith-Thomas (1982) presentan soluciones óptimas para los ejercicios 7.43 y 7.44. Knuth (1998) tiene el ejercicio 7.21.

Gibbons (1985) es un libro sobre teoría y algoritmos de grafos; cubre temas de este capítulo y muchos otros. Véase también Even (1973) y Even (1979); Aho, Hopcroft y Ullman (1974); Deo (1974); Reingold, Nievergelt y Deo (1977); y Sedgewick (1988).

8

Problemas de optimización de grafos y algoritmos codiciosos

8.1 Introducción

En este capítulo estudiaremos varios problemas de optimización de grafos que se pueden resolver exactamente empleando algoritmos codiciosos. Es común que en los problemas de optimización el algoritmo tenga que tomar una serie de decisiones cuyo efecto general es reducir al mínimo el costo total, o aumentar al máximo el beneficio total, de algún sistema. El método codicioso consiste en tomar las decisiones sucesivamente, de modo que cada decisión individual sea la mejor de acuerdo con algún criterio limitado "a corto plazo" cuya evaluación no sea demasiado costosa. Una vez tomada una decisión, no se podrá revertir, ni siquiera si más adelante se hace obvio que no fue una buena decisión. Por esta razón, los métodos codiciosos no necesariamente hallan la solución óptima exacta de muchos problemas. No obstante, en el caso de los problemas que estudiaremos en este capítulo, es posible demostrar que la estrategia codiciosa *apropiada* produce soluciones óptimas. En el capítulo 13 veremos problemas con los que estrategias codiciosas muy similares fracasan. En el capítulo 10 veremos otros problemas con los que las estrategias codiciosas fracasan.

Este capítulo presenta un algoritmo ideado por R. C. Prim para hallar un árbol abarcante mínimo en un grafo no dirigido, un algoritmo íntimamente relacionado con el anterior ideado por E. W. Dijkstra para hallar caminos más cortos en grafos dirigidos y no dirigidos, y un segundo algoritmo para hallar un árbol abarcante mínimo, que se debe a J. B. Kruskal. Los tres algoritmos emplean una cola de prioridad para seleccionar la mejor opción actual de entre un conjunto de opciones candidatas.

8.2 Algoritmo de árbol abarcante mínimo de Prim

El primer problema que estudiaremos es el de hallar un árbol abarcante mínimo para un grafo no dirigido, conectado y ponderado. En el caso de los grafos no conectados, la extensión natural del problema consiste en hallar un árbol abarcante mínimo para cada componente conectado. Ya vimos que los componentes conectados se pueden determinar en tiempo lineal (sección 7.4.2).

Los árboles abarcantes mínimos sólo tienen sentido en los grafos no dirigidos cuyas aristas están ponderadas, así que todas las referencias a "grafos" en esta sección son a grafos no dirigidos, y los pesos son siempre pesos de aristas. Recordemos que la notación $G = (V, E, W)$ significa que W es una función que asigna un peso a cada una de las aristas de E. Ésta es meramente la descripción matemática. En la implementación por lo regular no hay tal "función"; el peso de cada arista simplemente se almacena en la estructura de datos para esa arista.

8.2.1 Definición y ejemplos de árboles abarcantes mínimos

Definición 8.1 Árbol abarcante mínimo

Un *árbol abarcante* para un grafo no dirigido conectado $G = (V, E)$ es un subgrafo de G que es un árbol no dirigido y contiene todos los vértices de G. En un grafo ponderado $G = (V, E, W)$, el peso de un subgrafo es la suma de los pesos de las aristas incluidas en ese subgrafo. Un *árbol abarcante mínimo* (*MST*, por sus siglas en inglés) para un grafo ponderado es un árbol abarcante cuyo peso es mínimo. ■

Hay muchas situaciones en las que es preciso hallar árboles abarcantes mínimos. Siempre que se busca la forma más económica de conectar un conjunto de terminales, trátese de ciudades, terminales eléctricas, computadoras o fábricas, empleando por ejemplo carreteras, cables o líneas

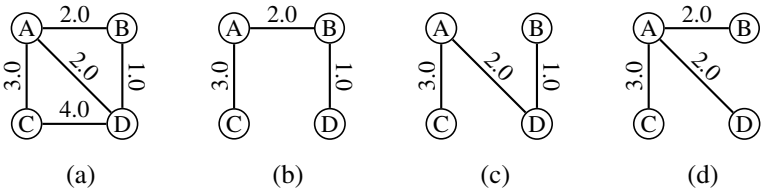


Figura 8.1 Grafo y algunos árboles abarcantes. Dos de ellos son *mínimos*

telefónicas, una solución es un árbol abarcante mínimo para el grafo que tiene una arista por cada posible conexión, ponderada con el costo de dicha conexión. Hallar árboles abarcantes mínimos también es un subproblema importante en diversos algoritmos de ruteo, es decir, algoritmos para hallar caminos eficientes a través de un grafo que visiten todos los vértices (o todas las aristas).

Como muestra el sencillo ejemplo de la figura 8.1, un grafo ponderado podría tener más de un árbol abarcante mínimo. De hecho, el método empleado para transformar un árbol abarcante mínimo en otro en este ejemplo es una ilustración de una propiedad general de los árboles abarcantes mínimos, que veremos en la sección 8.2.3.

8.2.2 Generalidades del algoritmo

Puesto que un árbol no dirigido está conectado y cualquier vértice se puede considerar como la raíz, una estrategia natural para hallar un árbol abarcante mínimo consiste en "cultivarlo" arista por arista a partir de algún vértice inicial. Quizá lo más conveniente sea probar primero nuestros métodos de recorrido estándar, búsqueda primero en profundidad y búsqueda primero en amplitud. Si podemos adaptar uno de estos esqueletos para resolver el problema, tendremos una solución en tiempo lineal, que seguramente es óptima. Recomendamos al lector dedicar cierto tiempo a probar algunas ideas empleando esos métodos de búsqueda, y construir grafos de ejemplo en los que no se pueda hallar el mínimo (véase el ejercicio 8.1).

Una vez convencidos de que un simple recorrido no es apropiado, y en vista de que se trata de un problema de optimización, la siguiente idea natural es probar el *método codicioso*. La idea básica del método codicioso es avanzar escogiendo una acción que incurra en el costo a corto plazo más bajo posible, con la esperanza de que muchos costos a corto plazo pequeños den un costo total también pequeño. (La posible desventaja es que acciones con costo a corto plazo bajo podrían llevar a una situación en la que no sea posible evitar costos altos posteriores.) Tenemos una forma muy natural de minimizar el costo a corto plazo de añadir una arista al árbol que estamos "cultivando": simplemente añadimos una arista que esté unida al árbol por exactamente un extremo y tenga el peso más bajo de todas las aristas que están en ese caso. El algoritmo de Prim adopta este enfoque codicioso.

Ahora que ya tenemos una idea de cómo resolver el problema, debemos hacernos las dos preguntas de siempre. ¿Funciona correctamente? ¿Con qué rapidez se ejecuta? Como ya hemos dicho, una serie de costos a corto plazo pequeños podría llevarnos a una situación poco favorable, así que, aunque tengamos la certeza de haber obtenido un árbol abarcante, falta ver si su peso es el mínimo de entre todos los árboles abarcantes. Además, puesto que necesitamos escoger entre muchas aristas en cada paso, y el conjunto de candidatas cambia después de cada decisión, es preciso pensar en qué estructuras de datos podrían hacer eficientes estas operaciones. Volveremos a estas preguntas después de precisar la idea general.

Lo primero que hace el algoritmo de Prim es seleccionar un vértice inicial arbitrario; luego se "ramifica" desde la parte del árbol que se ha construido hasta el momento escogiendo un nuevo vértice y arista en cada iteración. La nueva arista conecta al nuevo vértice con el árbol anterior. Durante la ejecución del algoritmo, podemos considerar que los vértices están divididos en tres categorías (disjuntas), a saber:

1. vértices *de árbol*: los que están en el árbol que se ha construido hasta ese momento,
2. vértices *de borde*: los que no están en el árbol, pero están adyacentes a algún vértice del árbol,
3. vértices *no vistos*: todos los demás.

El paso clave del algoritmo es la selección de un vértice del borde y una arista incidente. En realidad, puesto que los pesos están en las aristas, la decisión se concentra en la arista, no en el vértice. El algoritmo de Prim siempre escoge la arista de peso más bajo que va de un vértice de árbol a un vértice de borde. La estructura general del algoritmo podría describirse como sigue:

```
primMST(G, n) // BOSQUEJO
    Inicializar todos los vértices como no vistos.
    Seleccionar un vértice arbitrario s para iniciar el árbol; reclasificarlo como de árbol.
    Reclasificar todos los vértices adyacentes a s como de borde.
    Mientras haya vértices de borde:
        Seleccionar una arista con peso mínimo entre un vértice de árbol t y un vértice de
          borde v;
        Reclasificar v como de árbol; añadir la arista tv al árbol;
        Reclasificar todos los vértices no vistos adyacentes a v como de borde.
```

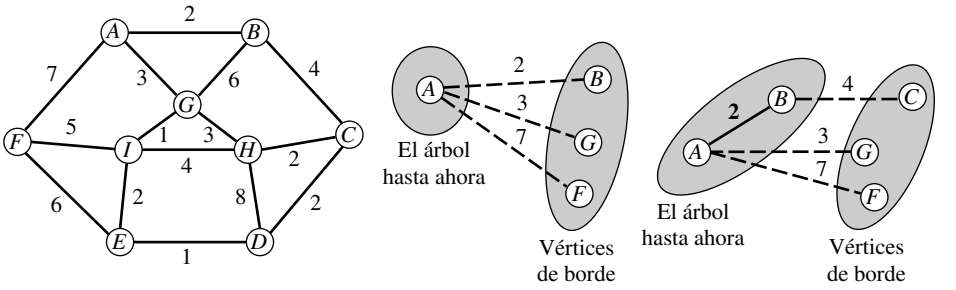
Ejemplo 8.1 Algoritmo de Prim, una iteración

La figura 8.2(a) muestra un grafo ponderado. Supóngase que *A* es el vértice inicial. Los pasos previos al ciclo nos llevan a la figura 8.2(b). En la primera iteración del ciclo, se determina que la arista de peso mínimo que conduce a un vértice de borde es *AB*. Por tanto, añadimos *B* al árbol y los vértices no vistos adyacentes a *B* entran en el borde; esto conduce a la figura 8.2(c). ■

¿Podemos tener la certeza de que esta estrategia produce un árbol abarcante mínimo? ¿Ser codiciosos a corto plazo es una buena estrategia a largo plazo? En este caso sí. En las dos subsecciones que siguen estudiaremos una propiedad general de todos los árboles abarcantes mínimos y la usaremos para demostrar que el árbol construido en cada etapa del algoritmo de Prim es un árbol abarcante mínimo en el subgrafo que ese árbol abarca. Volveremos a las consideraciones de implementación en la sección 8.2.5.

8.2.3 Propiedades de los árboles abarcantes mínimos

La figura 8.1 puso de manifiesto que un grafo ponderado puede tener más de un árbol abarcante mínimo. De hecho, los árboles abarcantes mínimos tienen una propiedad general que nos permite transformar cualquier árbol abarcante mínimo en otro siguiendo ciertos pasos. Examinar dicha propiedad también nos ayudará a familiarizarnos con los árboles no dirigidos en general.



(a) Un grafo ponderado

(b) El árbol y el borde después de escogerse el vértice inicial

(c) Después de seleccionar una arista y vértice: no se muestra *BG* porque *AG* es una mejor opción (tiene menor peso) para llegar a *G*

Figura 8.2 Una iteración del ciclo del algoritmo de Prim: las líneas continuas son aristas de árbol, y las punteadas, aristas a vértices de borde.

Definición 8.2 Propiedad de árbol abarcante mínimo

Sea $G = (V, E, W)$ un grafo ponderado conectado y sea T cualquier árbol abarcante de G. Supóngase que, para cada arista uv de G que *no está* en T, si uv se añade a T se crea un ciclo tal que uv es una arista de peso máximo de ese ciclo. En tal caso, el árbol T tiene la *propiedad de árbol abarcante mínimo* (*propiedad MST*, para abreviar). ■

Primero veremos qué significa la definición. Luego demostraremos que el nombre es apropiado; ¡el mero hecho de llamarla "propiedad de árbol abarcante mínimo" no implica que tenga algo que ver con los árboles abarcantes mínimos!

Ejemplo 8.2 Propiedad de árbol abarcante mínimo

Por definición, una arista no dirigida conecta cualesquier dos vértices de un árbol, y no tiene ciclos. Examinemos la figura 8.1, que muestra un grafo sencillo que llamaremos G y tres árboles abarcantes. Llamemos primero T al árbol de la parte (b). Supóngase que añadimos a T una arista de G que no está en T, formando un nuevo subgrafo G_1 (añadiremos la que pesa 2). Esto crea un ciclo (y sólo un ciclo) en el subgrafo G_1. (¿Por qué?) Ninguna otra arista del ciclo tiene un peso mayor que 2, que es el peso de la nueva arista. Como alternativa, si añadimos la arista que pesa 4, ninguna de las otras aristas del ciclo que se forma con *esa* arista pesará más de 4 (de hecho, ninguna pesa más de 3). No hay más aristas que probar, así que T tiene la propiedad de árbol abarcante mínimo. El árbol de la parte (c) es similar.

Sea ahora T el árbol de la parte (d), y añadamos la arista faltante que pesa 1. En esta ocasión, alguna otra arista del ciclo así formado pesa más de 1. Por tanto, este T *no* tiene la propiedad de árbol abarcante mínimo. Obsérvese que podemos extirpar cualquier arista de este ciclo para formar un nuevo subgrafo G_2, así que G_2 también debe ser un árbol (de hecho, un árbol abarcante). Esto se demuestra en el ejercicio 8.2 y se usa para demostrar el lema y el teorema siguientes. Pues-

to que existe una arista con peso mayor que 1, optamos por extirpar una de esas aristas. Ello implica que G_2 pesa menos que T, así que T no podría ser un árbol abarcante mínimo. ■

Lema 8.1 En un grafo ponderado conectado $G = (V, E, W)$, si T_1 y T_2 son árboles abarcantes que poseen la propiedad MST, tienen el mismo peso total.

Demostración La demostración es por inducción con k, el número de aristas que están en T_1 y no están en T_2. (Hay asimismo exactamente k aristas en T_2 que no están en T_1.) El caso base es $k = 0$; en este caso, T_1 y T_2 son idénticos, así que tienen el mismo peso.

Para $k > 0$, supóngase que el lema se cumple para los árboles que difieren en j aristas, donde $0 \leq j < k$. Sea uv una arista *de peso mínimo* que está en uno de los árboles T_1 o T_2, pero no en ambos. Supóngase $uv \in T_2$; el caso en que $uv \in T_1$ es simétrico. Consideremos el camino (único) que va de u a v en T_1: $w_0, w_1, \ldots, w_p$, donde $w_0 = u$, $w_p = v$ y $p \geq 2$. Este camino debe contener alguna arista que no está en T_2. (¿Por qué?) Sea $w_i w_{i+1}$ tal arista. Por la propiedad MST de T_1, $w_i w_{i+1}$ no puede tener un peso *mayor* que el peso de uv. Por el hecho de que uv se escogió como arista de peso mínimo de entre todas las aristas que difieren, $w_i w_{i+1}$ no puede pesar *menos* que uv. Por tanto, $W(w_i w_{i+1}) = W(uv)$. Añadimos uv a T_1, con lo que creamos un ciclo, y luego quitamos $w_i w_{i+1}$ con lo que rompemos ese ciclo y dejamos un nuevo árbol abarcante T_1' cuyo peso total es igual al de T_1. Sin embargo, T_1' y T_2 sólo difieren en $k - 1$ aristas, así que por la hipótesis inductiva tienen el mismo peso total. Por consiguiente, T_1 y T_2 tienen el mismo peso total. □

La demostración de este lema también nos muestra el método paso por paso para transformar cualquier árbol abarcante mínimo, T_1, en cualquier otro, T_2. Escogemos una arista de peso mínimo en T_2 que no esté en T_1; llamémosla uv. Examinamos el camino que conduce de u a v en T_1. En algún punto de ese camino habrá una arista con el mismo peso que uv, la cual no está en T_2; digamos que esa arista es xy (véase el ejercicio 8.3). Quitamos xy y añadimos uv. Esto nos acerca un paso más a T_2. Repetimos el paso hasta que los árboles coinciden.

Teorema 8.2 En un grafo ponderado conectado $G = (V, E, W)$, un árbol T es un árbol abarcante mínimo si y sólo si tiene la propiedad MST.

Demostración (Sólo si) Supóngase que T es un árbol abarcante mínimo para G. Supóngase que existe alguna arista uv que no está en T, tal que la adición de uv crea un ciclo en el que alguna *otra* arista xy tiene un peso $W(xy) > W(uv)$. Entonces, la eliminación de xy creará un nuevo árbol abarcante con un peso total menor que el de T, lo que contradice el supuesto de que T tenía peso mínimo.

(Si) Supóngase que T tiene la propiedad MST. Sea $T_{mín}$ *cualquier* árbol abarcante mínimo de G. Por la primera mitad del teorema, $T_{mín}$ tiene la propiedad MST. Por el lema 8.1, T tiene el mismo peso total que $T_{mín}$. □

8.2.4 Corrección del algoritmo MST de Prim

Ahora usaremos la propiedad MST para demostrar que el algoritmo de Prim construye un árbol abarcante mínimo. Esta demostración adopta una forma que se presenta con frecuencia al usar in-

ducción: la afirmación a demostrar por inducción es un poco más detallada que el teorema que nos interesa. Por ello, primero demostramos esa afirmación más detallada como *lema*. Luego el teorema simplemente extrae la parte interesante del lema. En este sentido, el teorema se parece mucho a la "envoltura" de un procedimiento recursivo, que describimos en la sección 3.2.2.

Lema 8.3 Sea $G = (V, E, W)$ un grafo ponderado conectado con $n = |V|$; sea T_k el árbol de k vértices construido por el algoritmo de Prim, para $k = 1, \ldots, n$; y sea G_k el subgrafo de G inducido por los vértices de T_k (es decir, uv es una arista de G_k si es una arista de G y tanto u como v están en T_k). Entonces T_k tiene la propiedad MST en G_k.

Demostración La demostración es por inducción con k. El caso base es $k = 1$. En este caso, G_1 y T_1 contienen el vértice inicial y ninguna arista, así que T_1 tiene la propiedad MST en G_1.

Para $k > 1$, suponemos que T_j tiene la propiedad MST en G_j para $1 \leq j < k$. Suponemos que el k-ésimo vértice que el algoritmo de Prim añadirá al árbol es v, y que las aristas entre v y los vértices de T_{k-1} son $u_1v, \ldots, u_dv$. Para concretar, supóngase que u_1v es la arista de peso más bajo de todas éstas que el algoritmo escoge. Necesitamos verificar que T_k tiene la propiedad MST. Es decir, si xy es cualquier arista de G_k que no está en T_k, necesitamos demostrar que xy tiene peso máximo en el ciclo que se crearía añadiendo xy a T_k. Si $x \neq v$ y $y \neq v$, quiere decir que xy también estaba en G_{k-1} pero no en T_{k-1}, así que por la hipótesis inductiva es máxima en el ciclo que se crea al añadírsele a T_{k-1}. Sin embargo, éste es el mismo ciclo en T_k, así que T_k tiene la propiedad MST en este caso. Falta demostrar que se posee la propiedad cuando xy es una de las aristas $u_2v, \ldots, u_dv$ (puesto que u_1v está en T_k). Si $d < 2$, ya terminamos, así que suponemos que no es el caso.

Podría ser útil consultar la figura 8.3 durante el resto de la demostración. Consideremos el camino de v a u_i en T_k para cualquier i, $1 \leq i \leq d$. Supóngase que alguna arista de este camino pesa *más* que u_iv, que a su vez pesa por lo menos lo mismo que u_1v. (Si no, se satisfaría la propiedad MST.) Específicamente, sea dicho camino $v, w_1, \ldots, w_p$, donde $w_1 = u_1$ y $w_p = u_i$. Entonces $w_1, \ldots, w_p$ es un camino en T_{k-1}. Sea w_aw_{a+1} la *primera* arista de este camino cuyo peso es mayor que $W(u_iv)$ y sea $w_{b-1}w_b$ la *última* arista de este camino cuyo peso es mayor que $W(u_iv)$ (posiblemente, $a + 1 = b$; véase la figura 8.3). Afirmamos que w_a y w_b no pueden existir en T_{k-1} si éste fue construido por el algoritmo de Prim. Supóngase que w_a se añadió al árbol antes que w_b. Entonces todas las aristas del camino que va de w_1 (que es u_1) a w_a se añadirían antes que w_aw_{a+1} y antes que $w_{b-1}w_b$, porque todas tienen menor peso, y u_1v también se habría añadido antes que cualquiera de las dos. Asimismo, si w_b se añadió al árbol antes que w_a, u_iv se habría añadido antes que w_aw_{a+1} y antes que $w_{b-1}w_b$. Sin embargo, ni u_1v ni u_iv están en T_{k-1}, así que ninguna arista del camino $w_1, \ldots, w_p$ pesa más de $W(u_iv)$, y queda establecida la propiedad MST para T_k. □

Teorema 8.4 El algoritmo de Prim produce un árbol abarcante mínimo.

Demostración En la terminología del lema 8.3, $G_n = G$ y T_n es la salida del algoritmo, de lo cual se sigue que T_n tiene la propiedad MST en G. Por el teorema 8.2, T_n es un árbol abarcante mínimo de G. □

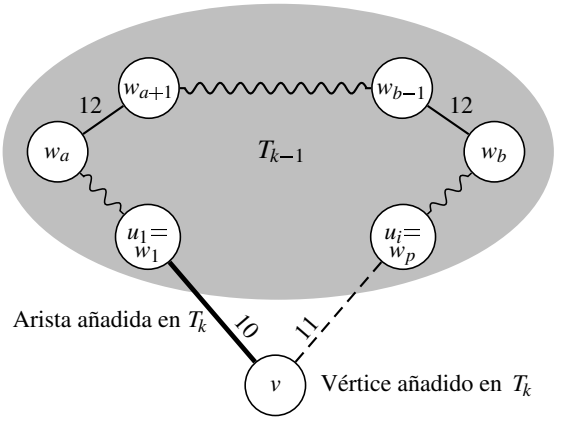


Figura 8.3 Ilustración para el lema 8.3. Los pesos que se dan son ejemplos. Las líneas onduladas son caminos en T_{k-1}. La arista punteada crearía un ciclo, como se muestra. Todas las aristas del camino entre u_1 ($= w_1$) y w_a, y las aristas del camino entre w_b y u_i ($= w_p$) tienen pesos no mayores que $W(u_i v)$, que es 11 en este ejemplo. Posiblemente, $w_b = w_{a+1}$ y $w_{b-1} = w_a$.

8.2.5 Cómo manejar el borde de manera eficiente con una cola de prioridad

Después de cada iteración del ciclo del algoritmo, podría haber nuevos vértices de borde, y el conjunto de aristas de entre las cuales se escogerá la siguiente cambiará. La figura 8.2(c) sugiere que no es necesario considerar todas las aristas entre vértices de árbol y vértices de borde. Después de escogerse *AB*, *BG* se convirtió en una opción, pero se desechó porque *AG* pesa menos y sería una mejor opción para llegar a *G*. Si *BG* pesara menos que *AG*, podría desecharse *AG*. Para cada vértice de borde, sólo necesitamos recordar una de las aristas que llegan a él desde el árbol: la de menor peso. Llamamos *aristas candidatas* a tales aristas.

El TDA de cola de prioridad (sección 2.5.1) tiene precisamente las operaciones que necesitamos para implementar el bosquejo del algoritmo dado en la sección 8.2.2. La operación `insertar` introduce un vértice en la cola de prioridad. La operación `obtenerMin` puede servir para escoger el vértice de borde que se puede conectar al árbol actual incurriendo en un costo mínimo. La operación `borrarMin` saca a ese vértice del borde. La operación `decrementarClave` registra un costo más favorable para la conexión de un vértice de borde cuando se descubre una mejor arista candidata. El costo mínimo conocido de conectar cualquier vértice de borde es el `pesoBorde` de ese vértice. Este valor hace las veces de prioridad del vértice y lo devuelve la función de acceso `obtenerPrioridad`.

Utilizando las operaciones del TDA de cola de prioridad, el algoritmo de alto nivel es el siguiente. Hemos introducido una subrutina `actualizarBorde` para procesar los vértices adyacentes al vértice seleccionado v. La figura 8.4 muestra un ejemplo de la acción del algoritmo.

```
primMST(G, n) // BOSQUEJO
    Inicializar la cola de prioridad cp como cola vacía.
    Seleccionar un vértice arbitrario s para iniciar el árbol;
    Asignar (−1, s, 0) a su arista candidata e invocar insertar(cp, s, 0).
    Mientras cp no esté vacía:
       v = obtenerMin(cp); borrarMin(cp);
       Añadir la arista candidata de v al árbol.
       actualizarBorde(cp, G, v);

actualizarBorde(cp, G, v) // BOSQUEJO
    Para todos los vértices w adyacentes a v, con nuevoPeso = W(v, w):
       Si w es no visto:
          Asignar (v, w, nuevoPeso) a su arista candidata.
          insertar(cp, w, pesoNuevo);
       Si no, si pesoNuevo < pesoBorde de w:
          Cambiar su arista candidata a (v, w, nuevoPeso).
          decrementarClave(cp, w, nuevoPeso);
```

Análisis preliminar

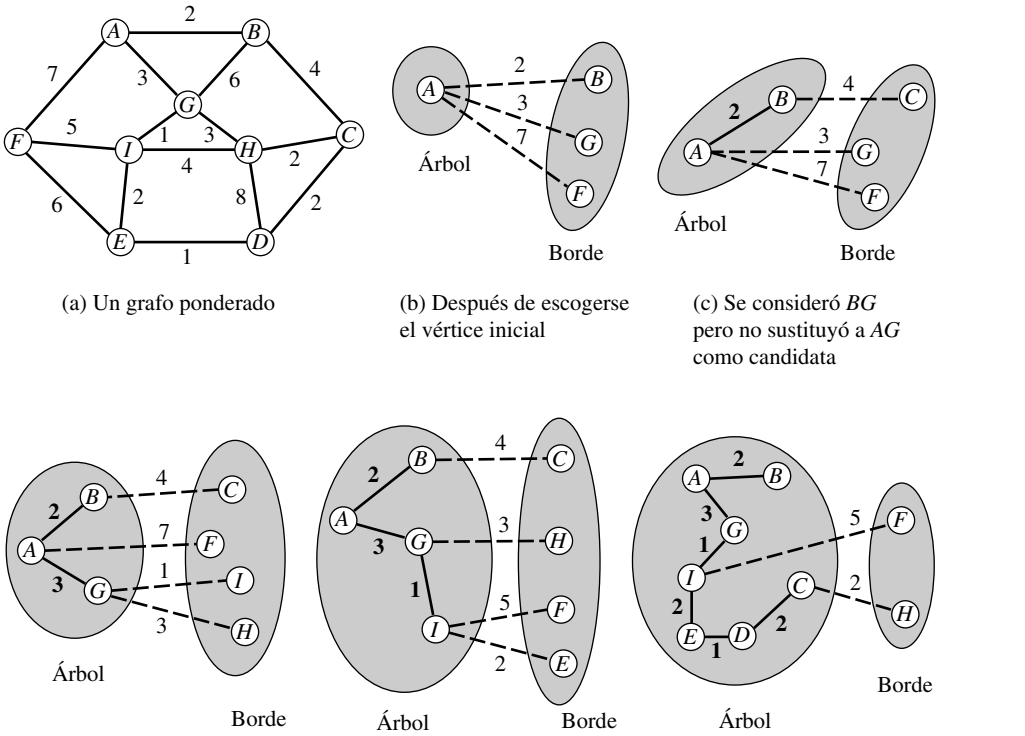
¿Qué podemos decir acerca del tiempo de ejecución de este algoritmo sin saber cómo se implementa el TDA de cola de prioridad? El primer paso sería estimar cuántas veces se efectúa cada operación del TDA. Luego podríamos escribir una expresión en la que los costos de las operaciones del TDA son parámetros. Supóngase, como es nuestra costumbre, que el grafo tiene n vértices y m aristas. Es fácil ver que el algoritmo ejecuta `insertar`, `obtenerMin` y `borrarMin` aproximadamente n veces, y que ejecuta `decrementarClave` cuando más m veces. (Hay $2m$ iteraciones del ciclo **`for`** que ejecuta `decrementarClave` porque cada arista se procesa desde ambas direcciones, pero veremos después que la condición del segundo **`if`** no se satisface más de una vez para cada arista.) Trabajando un poco podremos construir ejemplos en los que prácticamente cada arista dispare un `decrementarClave`. Es razonable suponer que `insertar` es menos costoso que `borrarMin`. Por tanto, tenemos una expresión en la que las T de la derecha denotan el tiempo medio que tarda la operación indicada en el curso de una ejecución del algoritmo:

$$T(n, m) = O\,(n\,T\,(\texttt{obtenerMin}) + n\,T\,(\texttt{borrarMin}) + m\,T\,(\texttt{decrementarClave})). \qquad (8.1)$$

En grafos generales m podría ser mucho mayor que n, por lo que obviamente nos interesa una implementación que se concentre en la eficiencia de `decrementarClave`.

Aquí nos damos cuenta de la ventaja de diseñar con tipos de datos abstractos. Ya razonamos que nuestro algoritmo es correcto, sea como sea que se implemente el TDA de cola de prioridad, siempre que la implementación cumpla con las especificaciones lógicas del TDA. Ahora estamos en libertad de adecuar la implementación a fin de minimizar, o por lo menos reducir, el miembro derecho de la ecuación (8.1).

Ya vimos que un montón es una implementación eficiente de una cola de prioridad (sección 4.8.1). ¿Cómo le va en la ecuación (8.1)? Esta pregunta es el tema del ejercicio 8.9, donde descubrimos que el peor caso es peor que $\Theta(n^2)$. ¿Podemos mejorarlo?



(a) Un grafo ponderado

(b) Después de escogerse el vértice inicial

(c) Se consideró *BG* pero no sustituyó a *AG* como candidata

(d) Después de seleccionarse *AG* y de actualizarse el borde y las candidatas

(e) *IF* ha sustituido a *AF* como candidata

(f) Después de varias pasadas más: las dos aristas candidatas se colocarán en el árbol

Figura 8.4 Un ejemplo de árbol abarcante mínimo de Prim

Si se quiere mejorar en general el tiempo del montón, es evidente que será preciso considerar implementaciones con las que `decrementarClave` se ejecute en un tiempo menor que $\Theta(\log n)$. Sin embargo, podemos darnos el lujo de hacer que `obtenerMin` y `borrarMin` tarden más de $\Theta(\log n)$ como concesión. ¿Hay alguna implementación en la cual `decrementarClave` tarde $O(1)$ y las otras operaciones no tarden más de $O(n)$?, si así fuera, la ecuación (8.1) daría $\Theta(n^2 + m) = \Theta(n^2)$. Invitamos a los lectores a considerar algunas alternativas antes de continuar.

■ ■ ■

La respuesta es tan simple, que es probable que la pasemos por alto. Basta con almacenar la información en uno o más arreglos, indizados por número de vértice. Es decir, podemos usar un arreglo aparte para cada campo, o juntar los campos en una clase organizadora y tener un arreglo cuyos elementos sean objetos de esa clase. Optaremos por usar arreglos individuales porque ello simplifica un poco la sintaxis. La operación `decrementarClave` es $O(1)$ porque basta con se-

guir el índice al vértice y actualizar dos o tres campos. La operación `obtenerMin` se efectúa examinando las *n* entradas de los arreglos; `borrarMin` puede usar el resultado del `obtenerMin` precedente o examinar otra vez los arreglos si el resultado anterior es obsoleto. Uno de los arreglos deberá ser el indicador de *situación* para indicar si el vértice está o no en el borde; sólo esos vértices serán elegibles como mínimos. Otro arreglo contiene la prioridad del vértice. En nuestro algoritmo, la prioridad del vértice debe corresponder siempre a `pesoBorde`, el peso de la arista candidata para ese vértice. Asimismo, las aristas candidatas se pueden mantener en un arreglo llamado `padre`, como se hizo el las búsquedas primero en amplitud y primero en profundidad. Es decir, `(v, padre[v])` es la arista candidata para *v*. Con esta implementación hemos llegado básicamente al algoritmo de Prim clásico.

Aunque el tema del encapsulamiento del diseño de TDA sugiere que las estructuras de datos de la cola de prioridad estén ocultas, las "desabstraeremos" para que todas las partes del algoritmo tengan acceso simple. No obstante, seguiremos usando las operaciones del TDA para *actualizar* esas estructuras de datos.

El algoritmo de Prim se publicó antes de que se inventaran las colas de prioridad y antes de que los lenguajes de programación apoyaran las estructuras de datos modernas, así que su implementación se basó simplemente en arreglos. Desde entonces, se han realizado investigaciones sustanciales acerca de la eficiencia de las colas de prioridad. Después de presentar la implementación más sencilla del algoritmo de Prim, mostraremos (en la sección 8.2.8) cómo adaptar la estructura de datos de *bosque de apareamiento* (sección 6.7.2) para este algoritmo. Esto puede servirnos como guía para usar otras implementaciones avanzadas de la cola de prioridad, como el *montón de Fibonacci*, que rebasan el alcance de este libro. (Véase Notas y referencias al final del capítulo.)

8.2.6 Implementación

Las principales estructuras de datos para el algoritmo (además de las empleadas para el grafo mismo) son tres arreglos: `situación`, `pesoBorde` y `padre`, indizados por número de vértice. La clasificación de los vértices está dada por el arreglo `situacion` y suponemos que hemos definido constantes con los nombres `novisto`, `deborde` y `dearbol`. Existe una fuerte correlación entre éstos y los colores blanco, gris y negro empleados en la búsqueda primero en amplitud (algoritmo 7.1). A veces, una búsqueda basada en la cola de prioridad en vez de la cola FIFO se denomina *búsqueda de primero el mejor*.

Los tres arreglos principales, `situacion`, `padre` y `pesoBorde` se reúnen en el objeto `cp` por comodidad al pasarlos a las subrutinas. Además, adecuaremos las operaciones `insertar` y `decrementarClave` de modo que registren el padre, no sólo la prioridad, del vértice que se está insertando o actualizando. Cuando se construye inicialmente `cp`, todos los elementos tienen la situación `novisto`.

Cuando invocamos `insertar` para un vértice, sus `padre` y `pesoBorde` adquieren valores y su situación cambia a `deborde`. Cabe señalar que `insertar` se invoca con el vértice inicial para crear el primer vértice de borde; su `padre` no es un vértice real.

Cuando invocamos `borrarMin`, la situación del vértice que actualmente es el mínimo cambia a `dearbol`, con lo que efectivamente lo sacamos de la cola de prioridad. Otro efecto de esto es que se congelan sus campos `pesoBorde` y `padre`.

En el ciclo principal del algoritmo `primMST`, conforme se recupera cada uno de los vértices (v) que en ese momento son el mínimo, su lista de adyacencia se procesa (en la subrutina `actualizarBorde`) para ver si alguna de estas aristas (v, w) ofrecen una conexión de menor costo con

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
`pesoBorde`	0	2	4			7	3	3	1
`padre`	−1	*A*	*B*			*A*	*A*	*G*	*G*
`situacion`	`dearbol`	`dearbol`	`deborde`	`novisto`	`novisto`	`deborde`	`dearbol`	`deborde`	`deborde`

Figura 8.5 Estructura de datos de árbol abarcante mínimo para la situación de la figura 8.4(d); no se muestran listas de adyacencia. Se supone que los vértices están en orden alfabético dentro de cada lista.

el vértice adyacente *w*. Suponemos que las listas de adyacencia contienen elementos de la clase organizadora `InfoArista` con dos campos, `a` y `peso`, según la descripción de la sección 7.2.3 y la ilustración de la figura 7.11.

La figura 8.5 muestra la estructura de datos en un punto intermedio de la ejecución del algoritmo del ejemplo de la figura 8.4 (específicamente, en el punto ilustrado por la figura 8.4d). Para facilitar su comprensión, mostramos los nombres de los vértices como letras, igual que en la figura 8.4.

Cuando el algoritmo termina, el arreglo `padre` implica las aristas del árbol. Es decir, para cada vértice *v* distinto del vértice inicial (raíz), `(v, padre[v])` es una arista del MST y `pesoBorde[v]` es su peso.

El contador `numCP` lleva la cuenta del número de vértices cuya situación es `deborde`, para que `estaVacia(cp)` se pueda ejecutar en tiempo constante. Si el grafo de entrada no está conectado, `padre[v]` y `pesoBorde[v]` no estarán definidos para los *v* que no estén conectados con el vértice inicial cuando el algoritmo termine. Esta condición nunca deberá presentarse porque una condición previa del algoritmo es que el grafo está conectado.

Algoritmo 8.1 Árbol abarcante mínimo (de Prim)

Entradas: Un arreglo `infoAdya` de listas de adyacencia que representa un grafo no dirigido, conectado y ponderado $G = (V, E, W)$, según se describe en la sección 7.2.3; n, el número de vértices; y s, el vértice inicial deseado. Todos los arreglos deberán estar definidos para los índices $1, \ldots, n$; el elemento número 0 no se usa. Los elementos de `infoAdya` son listas del TDA `ListaAristas` que se describe más adelante.

Salidas: Un árbol abarcante mínimo almacenado en el arreglo `padre` como árbol adentro y el arreglo `pesoBorde` que contiene, para cada vértice *v*, el peso de la arista entre `padre[v]` y *v*. (El padre de la raíz es −1.) El invocador reserva espacio para los arreglos y los pasa como parámetros y el algoritmo los llena.

Comentarios: El arreglo `situacion[1], ..., situacion[n]` denota la situación de búsqueda actual de todos los vértices. Los vértices no descubiertos son `novisto`; los que ya se descubrieron pero todavía no se procesan (en la cola de prioridad) son `deborde`; los que ya se procesaron son `dearbol`. Las listas de adyacencia son del tipo `ListaAristas` y tienen las operaciones estándar del TDA Lista (sección 2.3.2). Los elementos pertenecen a la clase organizadora `InfoArista` que tiene dos campos `a` y `peso`.

```
void primMST(ListaAristas[] infoAdya, int n, int s, int[] padre,
float[] pesoBorde)
    int[] situacion = new int[n+1];
    CPMin cp = crear(n, situacion, padre, pesoBorde);

    insertar(cp, s, −1, 0);
    while (estaVacia(cp) == false)
      int v = obtenerMin(cp);
      borrarMin(cp);
      actualizarBorde(cp, infoAdya[v], v);
    return;

/** Ver si se halla una mejor conexión con cualquier vértice
  * de la lista infoAdyaDeV, y decrementarClave en tal caso.
  * Para una conexión nueva, insertar el vértice. */
void actualizarBorde(CPMin cp, ListaAristas infoAdyaDeV, int v)
    ListaAristas adyaRest;
    adyaRest = infoAdyaDeV;
    while (adyaRest ≠ nil)
      InfoArista infoW = primero(adyaRest);
      int w = infoW.a;
      float nuevoPeso = infoW.peso;
      if (cp.situacion[w] == novisto)
        insertar(cp, w, v, nuevoPeso);
      else if (cp.situacion[w] == deborde)
        if (nuevoPeso < obtenerPrioridad(cp, w))
          decrementarClave(cp, w, v, nuevoPeso);
      adyaRest = resto(adyaRest);
    return;
```

La implementación de cola de prioridad se muestra en las figuras 8.6 a 8.8.

8.2.7 Análisis (tiempo y espacio)

Ahora completaremos el análisis del algoritmo 8.1 ejecutado con $G = (V, E, W)$. Nos habíamos quedado en la ecuación (8.1). El procedimiento principal es `primMST`, que invoca la subrutina `actualizarBorde`, pero ambos invocan las operaciones del TDA `CPMin`. Sean $n = |V|$ y $m = |E|$. El número de operaciones de inicialización (`crear`) es lineal en n. El cuerpo del ciclo **`while`** se ejecuta n veces, porque cada pasada efectúa un `borrarMin`. Necesitamos estimar el tiempo que se requiere para las invocaciones de procedimientos en este ciclo: `estaVacia`, `obtenerMin`, `borrarMin` y `actualizarBorde`.

Un parámetro de `actualizarBorde` es la lista `vertsAdya`, el cuerpo de su ciclo **`while`** se ejecuta una vez para cada elemento de esta lista (la invocación se omitiría para el último vértice en eliminarse de la cola de prioridad). Durante el curso del algoritmo, `actualizarBorde` procesa la lista de adyacencia de cada vértice una vez, así que el número total de pasadas por el cuerpo del ciclo **`while`** es aproximadamente $2m$. Una pasada por ese ciclo invoca `primero`, `resto` y

```
class CPMin
    // Campos de ejemplar
    int numVertices, numCP;
    int minVertice;
    float oo;
    int[] situacion;
    int[] padre;
    float[] pesoBorde;

    /** Construir cp con n vertices, todos "no vistos". */
    CPMin crear(int n, int[] situacion, int[] padre, float[] pesoBor-
    de)
      CPMin cp = new CPMin();
      cp.padre = padre;
      cp.pesoBorde = pesoBorde;
      cp.situacion = situacion;
      Inicializar situacion[1], ..., situacion[n] con novisto.
      cp.numVertices = n; cp.numCP = 0;
      cp.minVertice = -1;
      cp.oo = Float.POSITIVE_INFINITY;
      return cp;
```

Figura 8.6 Implementación de cola de prioridad para el algoritmo de árbol abarcante mínimo de Prim, parte 1: estructuras de datos y constructor del TDA. Los arreglos tienen un elemento por cada vértice del grafo; el elemento con índice 0 no se usa.

`obtenerPrioridad`, que suponemos están en $\Theta(1)$, pero también invoca `insertar` o `decrementarClave`. Con la implementación de la figura 8.7, `insertar` y `decrementarClave` también están en $\Theta(1)$, pero debemos tener presente que otras implementaciones quizá no lo logren; se trata de una decisión crucial: se podría invocar `decrementarClave` para casi todas las aristas de G, un total de aproximadamente $m - n$ invocaciones. Con la implementación que escogimos, el tiempo total para todas las invocaciones de `actualizarBorde` está en $\Theta(m)$.

Hasta aquí parece que el tiempo de ejecución del algoritmo podría ser lineal en m (G está conectado, así que m no puede ser mucho menor que n, pero podría ser mucho mayor). Sin embargo, `obtenerMin` se invoca aproximadamente n veces desde `primMST` y debe invocar la subrutina `hallarMin` en cada una de esas invocaciones. La subrutina `hallarMin` efectúa una comparación de peso para cada vértice que está "en" la cola de prioridad, a fin de hallar la arista candidata mínima. En el peor caso, no habrá vértices "no vistos" después de la primera invocación de `actualizarBorde`. Entonces el número medio de vértices que requerirán una comparación de peso será de aproximadamente $n/2$, puesto que se borra uno después de cada invocación de `obtenerMin`. (Nos concentramos en las comparaciones de pesos porque son inevitables; alguna otra implementación podría evitar la verificación de `situacion`.) Tenemos un total de (aproximadamente) $n^2/2$ comparaciones, aun si el número de aristas es menor. Una vez más, hacemos hincapié en que el tiempo que tarda `hallarMin` depende de la implementación y es una decisión

```
/** Asentar a nuevoPadre y nuevoP como padre y prioridad de v,
  * respectivamente, y hacer situacion[v] = deborde. */
void insertar(CPMin cp, int v, int nuevoPadre, float nuevoP)
    cp.padre[v] = nuevoPadre;
    cp.pesoBorde[v] = nuevoP;
    cp.situacion[v] = deborde;
    cp.verticeMin = -1;
    cp.numCP ++;
    return

/** Asentar nuevoPadre, nuevoP como padre, prioridad de v. */
void decrementarClave(CPMin cp, int v, int nuevoPadre, float nuevoP)
    cp.padre[v] = nuevoPadre;
    cp.pesoBorde[v] = nuevoP;
    cp.verticeMin = -1;
    return

/** Borrar de cp vértice de borde con peso mínimo. */
void borrarMin(CPMin cp)
    int viejoMin = obtenerMin(cp);

    cp.situacion[viejoMin] = dearbol;
    cp.verticeMin = -1;
    cp.numCP --;
    return
```

Figura 8.7 Implementación de cola de prioridad para el algoritmo de árbol abarcante mínimo de Prim, parte 2: procedimientos de manipulación.

crucial para la eficiencia general de `primMST`. Obsérvese que `borrarMin` también se invoca unas n veces, pero sólo requiere $O(1)$ por invocación; una implementación distinta podría desplazar el trabajo de `obtenerMin` a `borrarMin`. Por lo regular será necesario analizar juntas estas dos invocaciones.

Así pues, el tiempo de ejecución de peor caso, lo mismo que el número de comparaciones efectuadas en el peor caso, están en $\Theta(m + n^2) = \Theta(n^2)$. (Recomendamos al lector investigar formas de reducir el trabajo que se requiere para hallar las candidatas mínimas, pero véanse los ejercicios 8.7 a 8.9.)

La estructura de datos de la figura 8.5 emplea $3n$ celdas además de las de la representación del grafo con listas de adyacencia. Este espacio adicional es mayor que el ocupado por cualquiera de los algoritmos que hemos estudiado hasta ahora, y podría parecer demasiado. Sin embargo, hace posible una implementación del algoritmo eficiente en términos del tiempo. (Sería peor si el espacio extra requerido estuviera en $\Theta(m)$, pues para muchos grafos $\Theta(m) = \Theta(n^2)$.)

```
boolean estaVacia(CPMin cp)
    return (numCP == 0);

float obtenerPrioridad(CPMin cp, int v)
    return cp.pesoBorde[v];

/** Devolver vértice de borde con peso mínimo.
 ** Devolver -1 si no quedan vértices de borde.
  */
int obtenerMin(CPMin cp)
    if (cp.verticeMin == -1)
      hallarMin(cp);
    return cp.verticeMin;

// ¡Esta subrutina hace casi todo el trabajo!
void hallarMin(CPMin cp)
    int v;
    float pesoMin;

    pesoMin = cp.oo;
    for (v = 1; v <= cp.numVertices; v++)
      if (cp.situacion[v] == deborde)
        if (cp.pesoBorde[v] < pesoMin)
          cp.verticeMin = v;
          pesoMin = cp.pesoBorde[v];
      // Continuar ciclo
    return;
```

Figura 8.8 Implementación de cola de prioridad del algoritmo de árbol abarcante mínimo de Prim, parte 3: funciones de acceso y subrutina `hallarMin` de `obtenerMin`.

8.2.8 La interfaz de bosque de apareamiento

La estructura de datos de bosque de apareamiento general y las implementaciones de las operaciones de cola de prioridad se describieron en la sección 6.7.2. Adaptaciones menores permiten utilizarla en el algoritmo de Prim.

En primer lugar, la estructura de bosque de apareamiento general supone que un nodo de árbol contiene campos tanto para el identificador de elemento como para la prioridad. Sin embargo, en este caso el identificador (número de vértice) es suficiente porque podemos acceder a la prioridad de v como `pesoBorde[v]`. Por ello, los pasos de la sección 6.7.2 que dicen "crear `nuevoNodo` ..." deben modificarse para que digan "guardar la prioridad en `pesoBorde[v]` y crear `nuevoNodo` con `id` = v". El arreglo `refx` es un arreglo adicional parecido a `situacion`, el cual mantienen las operaciones del TDA de cola de prioridad. (Con algunos valores artificiales especiales de tipo `Arbol` para representar las situaciones `novisto` y `dearbol`, el arreglo `refx` puede sustituir al arreglo `situacion`, como optimización de espacio.) ¿Qué ganamos al usar el

bosque de apareamiento? Las operaciones `insertar` y `decrementarClave` se siguen ejecutando en tiempo constante. Los posibles ahorros están en la operación `obtenerMin`. En la implementación directa, `obtenerMin` debe examinar todo el arreglo `situacion` y posiblemente una buena parte del arreglo `pesoBorde` en cada operación. El bosque de apareamiento tiene la propiedad de que sólo las raíces de los árboles del bosque son candidatas para el mínimo. Aunque es difícil analizar cuántos árboles podría haber en diversos momentos durante la ejecución del algoritmo, es evidente que lo normal es que cada árbol tenga varios nodos, así que no será necesario examinar todos los vértices.

En general, el orden asintótico de peor caso no se conoce con exactitud. Sin embargo, se ha obtenido un resultado de optimidad parcial para una variante del TDA de bosque de apareamiento, llamada *montones de apareamiento de dos pasadas*. Para la clase de grafos en la que m crece según $\Theta(n^{1+c})$ para alguna constante $c > 0$, el costo amortizado de `obtenerMin` está en $\Theta(\log n)$ y los costos amortizados de `insertar` y `decrementarClave` están en $\Theta(1)$. Estas cotas implican que el algoritmo de Prim con montones de apareamiento de dos pasadas se ejecuta en $\Theta(m + n \log n) = \Theta(m)$ con esta clase de grafos.

Se sabe también que el algoritmo de Prim con montones de Fibonacci se ejecuta en $\Theta(m + n \log n)$ con todos los grafos, lo cual es asintóticamente óptimo. Sin embargo, se ha informado que los factores constantes para los montones de Fibonacci son muy grandes y las operaciones mismas son muy complicadas y difíciles de implementar. Por estas razones, se considera que el montón de apareamiento o el bosque de apareamiento es una alternativa práctica. El tema se trata en Notas y referencias al final del capítulo.

8.2.9 Cota inferior

¿Qué tanto trabajo es indispensable para hallar un árbol abarcante mínimo? Afirmamos que cualquier algoritmo de árbol abarcante mínimo requiere un tiempo en $\Omega(m)$ en el peor caso porque debe examinar, o procesar de alguna manera, todas las aristas del grafo. Para ver esto, sea G un grafo ponderado conectado en el que todas las aristas pesan por lo menos 2, y suponiendo que existiera un algoritmo que no hiciera absolutamente nada con una arista xy de G. Entonces xy no estaría en el árbol T que el algoritmo produce como salida. Cambiemos el peso de xy a 1. Esto no podría alterar la acción del algoritmo porque nunca examinó a xy. Sin embargo, ahora T no tiene la propiedad MST y, por el teorema 8.2, no es un árbol abarcante mínimo. De hecho, si queremos producir un árbol más ligero bastará con añadir xy a T para crear un ciclo, eliminando cualquier otra arista de ese ciclo. Por consiguiente, ningún algoritmo que no examine xy podrá ser correcto.

8.3 Caminos más cortos de origen único

En la sección 7.2 consideramos brevemente el problema de hallar la mejor ruta entre dos ciudades en un mapa de rutas aéreas, como la figura 7.8. Utilizando como criterio el precio de los pasajes de avión, observamos que la mejor forma —es decir, la más barata— de viajar de San Diego a Sacramento era haciendo una escala en Los Ángeles. Éste es un ejemplo, o aplicación, de un problema muy común en grafos ponderados; hallar un camino de peso mínimo entre dos vértices dados.

Da la casualidad que, en el peor caso, no es más fácil hallar un camino de peso mínimo entre un par de nodos s y t dado que hallar caminos de peso mínimo entre s y cualquier vértice al

que se puede llegar desde *s*. Este último problema se denomina *problema de camino más corto de origen único*. Se usa el mismo algoritmo para resolver ambos problemas.

En esta sección consideraremos el problema de hallar el camino de peso mínimo desde un vértice de origen dado hasta cualquier otro vértice de un grafo ponderado dirigido o no dirigido. El peso (longitud, o costo) de un camino es la suma de los pesos de las aristas que incluye. Si el peso se interpreta como distancia, decimos que un camino de peso mínimo es un *camino más corto*, y éste es el nombre que más a menudo se usa. (Lamentablemente, se acostumbra mezclar la terminología de peso, costo y longitud.)

¿Cómo determinamos el camino más corto de SD a SAC en la figura 7.8? De hecho, en ese ejemplo pequeño usamos un método muy poco algorítmico, plagado de supuestos (como el de que las tarifas eran proporcionales a la distancia entre las ciudades y el de que el mapa estaba dibujado aproximadamente a escala). Luego escogimos una ruta que "se veía" corta y calculamos su costo total. Por último, verificamos algunos otros caminos (de forma un tanto desorganizada) y no observamos ninguna mejora, así que declaramos el problema resuelto. Éste difícilmente sería un algoritmo que esperaríamos programar en una computadora. Lo mencionamos para contestar sinceramente la pregunta anterior; la gente por lo regular usa formas muy poco rigurosas de resolver problemas, sobre todo si el conjunto de datos es muy pequeño.

En la práctica, el problema de hallar caminos más cortos en un grafo se presenta en aplicaciones en las que *V* podría contener cientos, miles o incluso millones de vértices. En teoría, un algoritmo podría considerar todos los posibles caminos y comparar sus pesos, pero en la práctica ello podría requerir mucho tiempo, posiblemente siglos. Al tratar de hallar un mejor enfoque, es útil examinar algunas propiedades generales de los caminos más cortos para ver si sugieren una estrategia más eficiente. El algoritmo que presentamos se debe a E. W. Dijkstra y requiere que los pesos de las aristas no sean negativos. En Notas y referencias al final del capítulo se mencionan otros algoritmos que no imponen este requisito.

8.3.1 Propiedades de los caminos más cortos

En general, al tratar de resolver un problema grande, tratamos de dividirlo en problemas más pequeños. ¿Qué podemos decir acerca de los caminos más cortos entre nodos distantes, en términos de caminos más cortos entre nodos menos distantes? ¿Podemos usar algún enfoque del tipo de divide y vencerás? Supóngase que el camino *P* es un camino más corto de *x* a *y* y que *Q* es un camino más corto de *y* a *z*. ¿Implica esto que *P* seguido de *Q* es un camino más corto de *x* a *z*? No es difícil imaginar un ejemplo en el que lo anterior no es cierto. Sin embargo, hay una variación sutil de este tema que *sí* se cumple. La demostración de este lema se deja como ejercicio.

Lema 8.5 (Propiedad de camino más corto) En un grafo ponderado *G*, supóngase que un camino más corto de *x* a *z* consiste en un camino *P* de *x* a *y* seguido de un camino *Q* de *y* a *z*. En tal caso, *P* será un camino más corto de *x* a *y* y *Q* será un camino más corto de *y* a *z*. □

Supóngase que estamos tratando de hallar un camino más corto entre *x* y *z*. Quizá existe una arista directa *xz* que ofrece la ruta más corta. Sin embargo, si el camino más corto incluye dos o más aristas, el lema nos dice que se puede dividir en dos caminos, cada uno con menos aristas que el camino entero, cada uno de los cuales es un camino más corto por derecho propio. Si queremos desarrollar un algoritmo, necesitaremos establecer algún esquema organizado para desglosar caminos.

Ejemplo 8.3 Autobús lleno de turistas

Podemos entender mejor el problema si relacionamos los caminos más cortos desde un vértice de origen s con un proceso físico. Veamos el grafo como un grupo de islas conectadas por puentes de un solo sentido, como en el ejemplo 7.7, sólo que ahora los puentes tienen diferentes longitudes. La longitud del puente que corresponde a la arista uv es $W(uv)$.

Imaginemos un autobús lleno de turistas que deja a sus pasajeros en un vértice de origen s en el tiempo cero, igual que en ese ejemplo. Los turistas se dispersan desde s caminando a velocidad constante, digamos un metro por segundo. Cuando un grupo numeroso de turistas llega a una isla (vértice) nueva, se divide; grupos más pequeños toman cada uno de los puentes (aristas) que salen de esa isla. Es evidente que los primeros turistas en llegar a cualquier isla han seguido un camino más corto. En este ejemplo, "más corto" puede referirse al tiempo o a la distancia.

Consideremos la situación en la que los turistas están llegando por primera vez al vértice z. Supóngase que están recorriendo una arista yz. Entonces, un camino más corto de s a z pasa por y y, por el lema 8.5, consiste en un camino más corto de s a y seguido de la arista yz. ■

Simulación del grupo de turistas

Supóngase ahora que queremos *predecir* el momento en que llegarán los primeros turistas a z, y que tenemos un arreglo, `llegar`, para guardar los tiempos en que los primeros turistas llegan a cada vértice. Sean $y_1, y_2, \ldots, y_k$ los vértices conectados por una arista a z. El camino más corto tiene que usar uno de estos vértices, así que los consideraremos todos. Tan pronto como lleguen turistas a y_i, digamos en el tiempo `llegar`$[y_i]$, podremos predecir que los primeros turistas en llegar a z *no arribarán después de* `llegar`$[y_i] + W(y_iz)$. Por tanto, `llegar[z]` será el mínimo de esas predicciones. Puesto que no permitimos pesos negativos (nada de viajes al pasado para estos turistas), no tenemos que preocuparnos por los y_i desde los cuales los turistas lleguen después de `llegar[z]`.

¿Podemos usar una búsqueda primero en profundidad para organizar este cálculo? Puesto que necesitamos examinar los vértices de los cuales salen aristas *hacia* z, no las aristas que salen de z, nos interesa buscar en G^T, el grafo transpuesto (véase la definición 7.10). La idea general es que la búsqueda desde z iría a cada y_i, y al retroceder de y_i a z calcularíamos `llegar`$[y_i] + W(y_iz)$ y lo compararíamos con el valor previamente almacenado en `llegar[z]`. Cada vez que se obtenga un valor más pequeño, se guardará en `llegar[z]`. Exploraremos esta idea en el ejercicio 8.21, donde se demuestra que funciona con una clase importante de grafos, pero no con todos los grafos.

Otra idea natural para organizar el cálculo es el enfoque codicioso, puesto que hemos observado que podemos calcular `llegar[z]` una vez que conocemos los valores de `llegar`$[y_i]$ que son menores que `llegar[z]`. La heurística codiciosa en este caso consiste en hallar el vértice al cual los turistas llegarán *más pronto*, dado que ya han llegado a ciertos vértices.

8.3.2 Algoritmo de camino más corto de Dijkstra

En esta sección estudiaremos el algoritmo de camino más corto de Dijkstra; es muy similar en su enfoque y tiempos al algoritmo de árbol abarcante mínimo de Prim que vimos en la sección anterior.

Definición 8.3

Sea P un camino no vacío en un grafo ponderado $G = (V, E, W)$ que consta de k aristas xv_1, $v_1v_2, \ldots, v_{k-1}y$ (podría ser $v_1 = y$). El *peso* de P, denotado por $W(P)$, es la suma de los pesos

$W(xv_1)$, $W(v_1v_2)$, …, $W(v_{k-1}y)$. Si $x = y$, se considera que el camino vacío es un camino de x a y. El peso del camino vacío es cero.

Si ningún camino entre x y y pesa menos que $W(P)$, decimos que P es un *camino más corto*, o *camino de peso mínimo*. ■

Escogimos con cuidado las palabras de la definición anterior para dejar abierta la posibilidad de tener pesos negativos. No obstante, en esta sección supondremos que los pesos no son negativos. En tales circunstancias, los caminos más cortos pueden restringirse a los caminos simples.

Problema 8.1 Caminos más cortos de origen único

Se nos da un grafo ponderado $G = (V, E, W)$ y un vértice de origen s. El problema consiste en hallar un camino más corto de s a cada vértice v. ■

Antes de proceder, debemos considerar si necesitamos un algoritmo nuevo. Supóngase que usamos el algoritmo de árbol abarcante mínimo, partiendo de s. ¿El camino a v en el árbol construido por el algoritmo siempre será el camino más corto de s a v? Consideremos el camino de A a C en el árbol abarcante mínimo de la figura 8.4. Ése *no es* un camino más corto; el camino A, B, C es más corto.

El algoritmo de camino más corto de Dijkstra encuentra caminos más cortos de s a los demás vértices en orden de distancia creciente desde s. El algoritmo, al igual que el algoritmo MST de Prim de la sección 8.2, parte de un vértice (s) y se "ramifica" seleccionando ciertas aristas que conducen a nuevos vértices. El árbol construido por este algoritmo se denomina *árbol de caminos más cortos*. (Ponerle ese nombre no hace que sea cierto; hay que demostrar que los caminos del árbol realmente son caminos más cortos.)

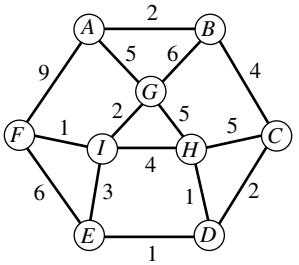
Otra semejanza con el algoritmo MST de Prim es que el algoritmo de Dijkstra es codicioso; siempre escoge una arista al vértice que parece estar "más cerca", aunque en este caso el sentido de "más cerca" es "más cerca de s", no "más cerca del árbol". Una vez más, los vértices se dividen en las tres categorías (disjuntas) siguientes:

1. vértices *de árbol*: las que están en el árbol construido hasta ahora,
2. vértices *de borde*: los que no están en el árbol, pero están adyacentes a algún vértice del árbol,
3. vértices *no vistos*: todos los demás.

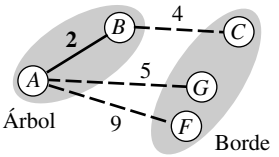
Además, como en el algoritmo de Prim, sólo recordamos una arista candidata (la mejor hallada hasta el momento) por cada vértice de borde. Para cada vértice de borde z hay por lo menos un vértice de árbol v tal que vz es una arista de G. Para cada v semejante hay un camino (único) de s a v en el árbol (podría ser que $s = v$); $d(s, v)$ denota el peso de ese camino. La adición de la arista vz a a este camino da un camino de s a z, y su peso es $d(s, v) + W(vz)$. La arista candidata para z es la arista vz tal que $d(s, v) + W(vz)$ sea mínimo entre todas las opciones de vértice v del árbol que se ha construido hasta ese momento.

Ejemplo 8.4 Crecimiento de un árbol de caminos más cortos

Examinemos el grafo de la figura 8.9(a). Cada arista no dirigida se trata como un par de aristas dirigidas en direcciones opuestas. Supóngase que el vértice de origen es A. Sigamos los pasos del

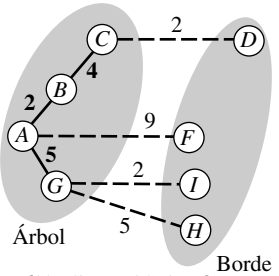


(a) El grafo



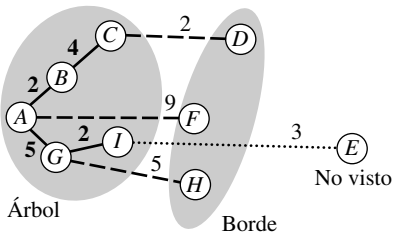
$d(A, B) + W(BC) = 6$
$d(A, A) + W(AG) = 5$
$d(A, A) + W(AF) = 9$
Escoger AG después

(b) Paso intermedio

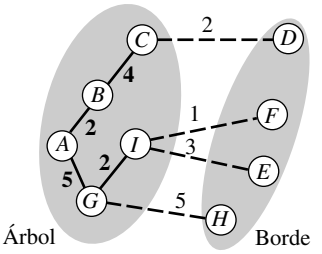


$d(A, C) + W(CD) = 8$
$d(A, A) + W(AF) = 9$
$d(A, G) + W(GI) = 7$
$d(A, G) + W(GH) = 10$
Seleccionar GI después

(c) Paso intermedio: se consideró CH para sustituir a GH como candidata, pero no se escogió



(d) Se seleccionó GI



(e) AF fue sustituida por IF como candidata

Figura 8.9 Ejemplo para el algoritmo de camino más corto de Dijkstra: el problema consiste en hallar el camino más corto de A a H.

"crecimiento" del árbol. En un principio, sólo tiene el vértice A, y $d(A, A) = 0$. Los turistas se bajan del autobús en A en el tiempo 0 y comienzan a caminar por los puentes AB, AG y AF. Al igual que en el ejemplo 7.7, los primeros turistas que llegan a una isla compran todas las gangas. Una vez que un grupo de turistas llegue tarde a una isla, sus integrantes ya no podrán ser los primeros en llegar a ninguna isla futura, porque el grupo anterior se dividió y exploró todos los puentes que salen de la isla. El algoritmo no seguirá la pista a estos grupos.

En la parte (b) de la figura se ha añadido la arista AB porque B es el vértice más cercano al origen A, y $d(A, B) = 2$. Un grupo de turistas llegó a B en el tiempo 2 y se dividió en subgrupos que comienzan a caminar por los puentes BA, BC y BG. Los demás grupos siguen caminando por AG y AF.

Ahora todos los vértices a los que se puede llegar por una arista desde A o desde B están en el borde, a menos que ya estén en el árbol, claro. Para cada vértice de borde, la arista candidata se muestra como línea punteada; obsérvese que BG no es una arista candidata. Sabemos que llegarán turistas a G a más tardar en el tiempo 5 (desde A), así que los que están cruzando el puente BG no van a ser los primeros, por tanto podemos olvidarnos de ellos.

Con base en las aristas de árbol y candidatas, G es el vértice de borde más cercano a A, así que AG será la siguiente arista en añadirse al árbol, y $d(A, G) = 5$. Es decir, llegan turistas a G en el tiempo 5 y se dividen en grupos para explorar GA, GB, GH y GI. El algoritmo sólo sigue la pista a los que se dirigen hacia H (pronóstico de llegada = 10) e I (pronóstico de llegada = 7). Sin embargo, los turistas que están en el puente BC llegan a C en el tiempo 6, así que la siguiente arista añadida será BC, y $d(A, C) = 6$. Salen turistas de C en el tiempo 6 y toman los puentes CB, CD y CH. Podemos olvidarnos de los que tomaron CB porque B ya se visitó, y también de los que tomaron CH porque llegarán a H en el tiempo 11, que es posterior al pronóstico de llegada de los turistas "competidores" que están en GH. Esto nos lleva a la parte (c) de la figura.

Dada la situación de la figura 8.9(c), el siguiente paso consiste en escoger una arista candidata y un vértice de borde. Escogemos como candidata a yz para la cual $d(s, y) + W(yz)$ es mínimo. Éste es el peso del camino que se obtiene juntando yz al camino conocido (y, es de esperar, más corto) de s a y. El vértice z se selecciona entre D, F, I y H, los vértices de borde actuales. En este caso, GI es la arista escogida, y $d(A, I) = 7$. ■

La estructura general del algoritmo de Dijkstra puede describirse así:

```
dijsktraCMCOU(G, n) // BOSQUEJO
    Inicializar todos los vértices como no vistos.
    Iniciar el árbol con el vértice de origen especificado s; reclasificarlo como de árbol;
    definir d(s, s) = 0.
    Reclasificar todos los vértices adyacentes a s como de borde.
    Mientras haya vértices de borde:
        Seleccionar una arista entre un vértice de árbol t y un vértice de borde v tal que
        (d(s, t) + W(tv)) sea mínimo;
        Reclasificar v como de árbol; añadir la arista tv al árbol;
        definir d(s, v) = (d(s, t) + W(tv)).
        Reclasificar todos los vértices no vistos adyacentes a v como de borde.
```

Puesto que la cantidad $d(s, y) + W(yz)$ para una arista candidata yz podría usarse varias veces, se le puede calcular una vez y guardar. Para calcularla de manera eficiente recién que yz se convierte en candidata, también guardamos $d(s, y)$ para cada y del árbol. Así, podríamos usar un arreglo `dist`, a saber:

`dist`$[y] = d(s, y)$ para y en el árbol;

`dist`$[z] = d(s, y) + W(yz)$ para z en el borde, donde yz es la arista candidata a z.

Al igual que en el algoritmo de Prim, una vez que se seleccionan un vértice y la candidata correspondiente, se hace necesario actualizar la información de algunos vértices de borde y hasta entonces no vistos en la estructura de datos.

Ejemplo 8.5 Actualización de información de distancia

En la figura 8.9(d) se acaban de seleccionar el vértice *I* y la arista *GI*. La arista candidata para *F* era *AF* (con `dist[F]` = 9), pero ahora es preciso sustituir *AF* por *IF* porque *IF* ofrece un camino más corto a *F*. También es preciso recalcular `dist[F]`. Por otra parte, *IH* no ofrece un camino más corto a *H* porque actualmente `dist[H]` = 10, así que no se volverá a considerar esta arista. El vértice *E*, que no se había visto, ahora está en el borde porque está adyacente a *I*, que ahora está en el árbol. La arista *IE* se convierte en candidata. Estos cambios dan pie a la figura 8.9(e). Es necesario calcular los valores de `dist` para los nuevos vértices de borde. ■

¿Funciona este método? El paso crucial es la selección del siguiente vértice de borde y la arista candidata. Para una candidata yz arbitraria, $d(s, y) + W(yz)$ no es necesariamente la distancia más corta de s a z, porque es posible que los caminos más cortos a z no pasen por y. (En la figura 8.9, por ejemplo, el camino más corto a *H* no pasa por *G*, aunque *GH* es una candidata en las partes c, d y e.) Afirmamos que, si $d(s, y)$ es la distancia más corta para cada vértice de árbol y, y yz se escoge de modo que $d(s, y) + W(yz)$ sea mínimo entre todas las candidatas, yz sí ofrece un camino más corto. Demostraremos esa afirmación en el teorema siguiente.

Teorema 8.6 Sea $G = (V, E, W)$ un grafo ponderado con pesos no negativos. Sea V' un subconjunto de V y sea s un miembro de V'. Supóngase que $d(s, y)$ es la distancia más corta de s a y en G, para cada $y \in V'$. Si se escoge la arista yz de modo que $d(s, y) + W(yz)$ entre todas las aristas que tienen un vértice y en V' y un vértice z en $V - V'$, entonces el camino que consiste en un camino más corto de s a y seguido de la arista yz es el camino más corto de s a z.

Demostración Véase la figura 8.10. Supóngase que se escoge $e = yz$ como se indica, y sea $s, x_1, \ldots, x_r, y$ un camino más corto de s a y (podría ser que $y = s$). Sea $P = s, x_1, \ldots, x_r, y, z$. $W(P) = d(s, y) + W(yz)$. Sea $s, z_1, \ldots, z_a, \ldots, z$ cualquier camino más corto de s a z; llamémoslo P'. Se escoge el vértice z_a de modo que sea el primer vértice de P' que no está en V' (podría ser que $z_a = z$). Debemos demostrar que $W(P) \leq W(P')$. (En el algoritmo, $z_{a-1}z_a$ sería una arista candidata; si $a = 1$, $z_0 = s$.) Por la e escogida,

$$W(P) = d(s, y) + W(e) \leq d(s, z_{a-1}) + W(z_{a-1}za). \tag{8.2}$$

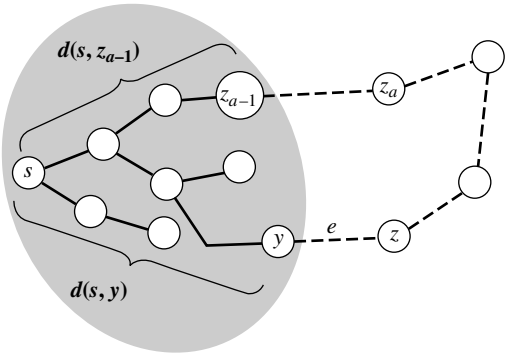


Figura 8.10 Para la demostración del teorema 8.6

Por el lema 8.5, $s, z_1, \ldots, z_{a-1}$ es un camino más corto de s a z_{a-1}, así que el peso de este camino es $d(s, z_{a-1})$. Puesto que $s, z_1, \ldots, z_{a-1}, z_a$ forma parte del camino P' y cualesquier aristas restantes deben tener peso no negativo,

$$d(s, z_{a-1}) + W(z_{a-1}z_a) \leq W(P'). \tag{8.3}$$

Combinando las ecuaciones (8.2) y (8.3), $W(P) \leq W(P')$. □

Teorema 8.7 Dado un grafo ponderado dirigido G con pesos no negativos y un vértice de origen s, el algoritmo de Dijkstra calcula la distancia más corta (peso de un camino de peso mínimo) de s a cada uno de los vértices de G a los que se puede llegar desde s.

Demostración La demostración es por inducción con el orden en que se añaden vértices al árbol de caminos más cortos. Los detalles se dejan para el ejercicio 8.16. □

8.3.3 Implementación

El algoritmo de camino más corto puede usar exactamente el mismo TDA de cola de prioridad que el algoritmo de Prim; véanse las figuras 8.6 a 8.8. Cuando el algoritmo termina, el arreglo `padre` implica las aristas del árbol de caminos más cortos. Es decir, para cada vértice v distinto del vértice de origen, (`v, padre[v]`) es una arista del árbol de caminos más cortos y `pesoBorde[v]` es la distancia de s a v. Si no es posible llegar a todos los vértices desde el origen dado, s, `padre[v]` y `pesoBorde[v]` no estarán definidos para los v a los que no se pueda llegar desde s. Es fácil ajustar el algoritmo para asignar a esos elementos valores especiales, como $n + 1$ e ∞.

Algoritmo 8.2 Caminos más cortos de origen único (de Dijkstra)

Entradas: Un arreglo `infoAdya` de listas de adyacencia que representan un grafo ponderado, dirigido o no dirigido, $G = (V, E, W)$, según la descripción de la sección 7.2.3; n, el número de vértices; y s, el vértice inicial deseado. Todos los arreglos deben estar definidos para los índices $1, \ldots, n$; el elemento con índice 0 no se usa. Los elementos de `infoAdya` son listas del TDA `ListaAristas`, que se describe más adelante.

Salidas: Un árbol de caminos más cortos, almacenado en el arreglo `padre` como árbol adentro, y el arreglo `pesoBorde` que contiene, para cada vértice v, la distancia más corta de s a v. (El padre de la raíz es -1.) El invocador reserva espacio para los arreglo y los pasa como parámetros, y el algoritmo se encarga de llenarlos.

Comentarios: El arreglo `situacion[1]`, ..., `situacion[n]` denota la situación de búsqueda actual de todos los vértices. Los vértices no descubiertos son `novisto`; los que ya se descubrieron pero aún no se procesan (en la cola de prioridad) son `deborde`; los que ya se procesaron son `dearbol`. Las listas de adyacencia son del tipo `ListaAristas` y tienen las operaciones estándar del TDA Lista (sección 2.3.2). Los elementos pertenecen a la clase organizadora `InfoArista` y tienen dos campos, `a` y `peso`.

```
void caminosMasCortos(ListaAristas[] infoAdya, int n, int s, int[]
    padre, float[] pesoBorde)
    int[] situacion = new int[n+1];
    CPMin cp = crear(n, situacion, padre, pesoBorde);

    insertar(cp, s, -1, 0);
    while (estaVacia(cp) == false)
      int v = obtenerMin(cp);
      borrarMin(cp);
      actualizarBorde(cp, infoAdya[v], v);
    return;

/** Ver si se encuentra una mejor conexión con cualquier vértice de
  * la lista infoAdyaDeV, y decrementarClave en tal caso.
  * Si la conexión es nueva, insertar el vértice. */
void actualizarBorde(CPMin cp, ListaAristas infoAdyaDeV, int v)
    float miDist = cp.pesoBorde[v];
    ListaAristas adyaRest;
    adyaRest = infoAdyaDeV;
    while (adyaRest ≠ nil)
      InfoArista infoW = primero(adyaRest);
      int w = infoW.a;
      float nuevaDist = miDist + infoW.peso;
      if (cp.situacion[w] == novisto)
        insertar(cp, w, v, nuevaDist);
      else if (cp.situacion[w] == deborde)
        if (nuevaDist < obtenerPrioridad(cp, w))
          decrementarClave(cp, w, v, nuevaDist);
      adyaRest = resto(adyaRest);
    return;
```

Análisis

El análisis efectuado en la sección 8.2.7 del algoritmo de árbol abarcante mínimo de Prim, algoritmo 8.1, también es válido para el algoritmo de caminos más cortos de Dijkstra, algoritmo 8.2, sin que sea necesario modificarlo. El algoritmo de Dijkstra también se ejecuta en un tiempo $\Theta(n^2)$ en el peor caso. También son válidas la cota inferior de $\Omega(m)$ y las necesidades de espacio de $\Theta(n)$.

Si se espera no poder llegar a un número apreciable de vértices, podría ser más eficiente incluir, como paso de procesamiento previo, una prueba de "asequibilidad", eliminar los vértices a los que no se puede llegar y renumerar los vértices restantes como $1, \ldots, n_r$. El costo total estaría en $\Theta(m + n_r^2)$, en vez de $\Theta(n^2)$.

Es posible usar el bosque de apareamiento (sección 6.7.2), el montón de apareamiento o el montón de Fibonacci para implementar la cola de prioridad en el algoritmo de Dijkstra, de forma análoga a la que se describió en la sección 8.2.8 para el algoritmo de Prim. Las cotas asintóticas son las mismas: el uso de un montón de Fibonacci da el orden asintótico óptimo de $\Theta(m + n \log n)$, pero presenta dificultades prácticas.

8.4 Algoritmo de árbol abarcante mínimo de Kruskal

Sea $G = (V, E, W)$ un grafo ponderado no dirigido. En la sección 8.2 estudiamos el algoritmo de Prim para hallar un árbol abarcante mínimo para G (con la condición de que G estuviera conectado). El algoritmo iniciaba en un vértice arbitrario y se ramificaba desde él escogiendo "codiciosamente" aristas de peso bajo. En cualquier momento, las aristas escogidas formaban un árbol. Aquí examinaremos un algoritmo que usa una estrategia más codiciosa aún. En toda esta sección, los grafos son grafos no dirigidos.

8.4.1 El algoritmo

El bosquejo general del algoritmo de Kruskal es el siguiente. En cada paso se escoge la arista restante de peso más bajo de cualquier punto del grafo, aunque se desecha cualquier arista que formaría un ciclo con las que ya se escogieron. En cualquier momento, las aristas escogidas hasta entonces forman un bosque, pero no necesariamente un árbol. El algoritmo termina cuando se han procesado todas las aristas.

```
kruskalMST(G, n) // BOSQUEJO
    R = E;   // R es las aristas restantes.
    F = ∅;   // F es las aristas de bosque.
    while (R no está vacío)
        Quitar la arista más ligera (más corta), vw, de R;
        if (vw no forma un ciclo en F)
            Añadir vw a F;
    return F;
```

Antes de siquiera pensar en cómo implementar esta idea, debemos preguntarnos si funciona. Puesto que el grafo podría no estar conectado, necesitamos primero una definición.

Definición 8.4 Colección de árboles abarcantes

Sea $G = (V, E, W)$ un grafo ponderado no dirigido. Una *colección de árboles abarcantes* para G es un conjunto de árboles, uno por cada componente conectado de G, tal que cada árbol es un árbol abarcante para su componente conectado. Una *colección de árboles abarcantes mínima* es una colección de árboles abarcantes cuyas aristas tienen un peso total mínimo, es decir, una colección de árboles abarcantes mínimos. ■

En primer lugar, ¿cada uno de los vértices de G está representado en algún árbol? (Podría haber varios árboles si el grafo no está conectado.) Sea v un vértice arbitrario de G. Si por lo menos una arista incide en v, la primer arista que se saque de R y que incida en v se incluirá en F. Por otra parte, si v es un vértice aislado (sin aristas incidentes), *no* estará representado en F y se le tendrá que considerar por separado para no pasarlo por alto.

La siguiente pregunta es si el algoritmo crea o no una colección de árboles abarcantes, suponiendo que G no tiene vértices aislados. Es decir, ¿hay exactamente un árbol en F para cada uno de los componentes conectados de G? El lema que sigue nos ayuda a entender esta pregunta. La demostración es fácil y se deja como ejercicio.

Lema 8.8 Sea F un bosque; es decir, cualquier grafo acíclico no dirigido. Sea $e = vw$ una arista que no está en F. Existe un ciclo que consta de e y aristas de F si y sólo si v y w están en el mismo componente conectado de F. □

Supóngase ahora que algún componente conectado de G corresponde a dos o más árboles en el bosque F que el algoritmo de Kruskal calcula. Deberá haber alguna arista en G que conecte dos de esos árboles, llamémosla vw; es decir, v y w están en diferentes componentes conectados de F. Por tanto, cuando el algoritmo procesó vw, debió haber formado un ciclo en el bosque en ese momento (llamémosle F') porque vw no se añadió a F'. Por el lema 8.8, v y w estaban en el mismo componente conectado de F en ese momento. Pero entonces es imposible que v y w estén en diferentes componentes conectados de F cuando el algoritmo termina. Por tanto, F sólo puede contener un árbol por cada componente conectado de G.

Habiendo determinado que el algoritmo calcula *alguna* colección de árboles abarcantes, la última pregunta en materia de corrección es si los árboles tienen peso mínimo o no. Esta pregunta se contesta con el teorema siguiente, cuya demostración se deja como ejercicio.

Teorema 8.9 Sea $G = (V, E, W)$ un grafo ponderado no dirigido. Sea $F \subseteq E$. Si F está contenido en una colección de árboles abarcantes mínima para G, y si e es una arista de peso mínimo en $E - F$ tal que $F \cup \{e\}$ no tiene ciclos, entonces $F \cup \{e\}$ está contenido en una colección de árboles abarcantes mínima para G. □

El algoritmo inicia con $F = \varnothing$ y agrega aristas a F hasta que se han procesado todas las aristas. El teorema garantiza que F siempre está contenido dentro de alguna colección de árboles abarcantes mínima, y ya nos dimos cuenta de que el valor final de F es una colección de árboles abarcantes para G, con la excepción de los árboles triviales que consisten en nodos aislados sin aristas.

Ya estamos en condiciones de considerar métodos de implementación. Para acceder a las aristas en orden de peso creciente, usamos una cola de prioridad minimizante (sección 2.5.1), como un montón (sección 4.8.1). Las aristas de F se pueden almacenar en una lista, pila u otra estructura de datos conveniente.

Un problema que es preciso resolver es cómo determinar si una arista formará un ciclo con otras que ya están en F. El lema 8.8 proporciona el criterio: si v y w están en el mismo componente conectado de F, entonces (y sólo en este caso) la adición de la arista vw a F creará un ciclo. Por ello, nos conviene ir recordando los componentes conectados de F conforme se construye. En particular, dados dos vértices v y w, queremos poder determinar de manera eficiente si están o no

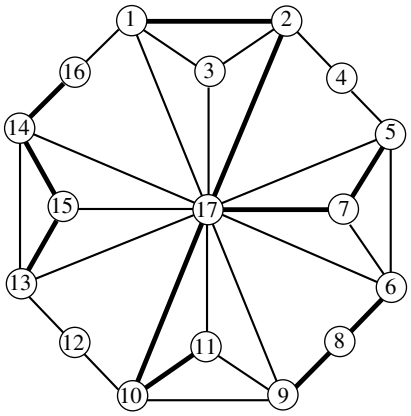


Figura 8.11 Las aristas gruesas están en el subgrafo *F*. Las clases de equivalencia son {1, 2, 5, 7, 10, 11, 17}, {6, 8, 9}, {13, 14, 15, 16}, {3}, {4} y {12}.

en el mismo componente conectado de *F*. Podemos aplicar la metodología de relaciones de equivalencia dinámicas que desarrollamos en la sección 6.6.

Definimos una relación, "≡", entre los vértices de un subgrafo *F* como $v \equiv w$ si y sólo si v y w están en el mismo componente conectado de *F*. Es fácil verificar que ≡ es una relación de equivalencia. (Véase un ejemplo en la figura 8.11.) Así pues, por el lema 8.8, el algoritmo de Kruskal escoge una arista vw si y sólo si $v \not\equiv w$. En un principio, todos los vértices de *G* están en la relación ≡ como clases de equivalencia individuales, y *F* es un grafo que consiste en todos los vértices de *G*, pero ninguna arista (con esto también damos cuenta de los vértices aislados). Cada vez que se escoge una arista, el subgrafo *F* y la relación de equivalencia ≡ cambian; cada arista nueva hace que dos componentes conectados, o dos clases de equivalencia, se fusionen en uno solo.

El mantenimiento y consulta de la relación ≡ se efectúan mediante el TDA Unión-Hallar. Recordemos que `hallar`(v) devuelve el identificador único de la clase de equivalencia del vértice v, y que si s y t son identificadores de clases de equivalencia distintas, `union`(s, t) las fusiona.

Algoritmo 8.3 Árbol abarcante mínimo (de Kruskal)

Entradas: $G = (V, E, W)$, un grafo ponderado, con $|V| = n$, $|E| = m$.

Salidas: *F*, un subconjunto de *E* que forma un árbol abarcante mínimo para *G*, o una colección de árboles abarcantes mínima si *G* no está conectado.

Comentarios: La estructura `conjuntos` definida en el algoritmo corresponde a la relación de equivalencia ≡ de la explicación. Omitimos los calificadores de nombre de clase en las operaciones del TDA Unión-Hallar y del TDA de cola de prioridad para hacerlas más comprensibles.

```
kruskalMST(G, n, F) // BOSQUEJO
    int cuenta;
    Construir una cola de prioridad minimizante, cp, de aristas de G, en la que las prioridades son los pesos.
    Inicializar una estructura Unión-Hallar, conjuntos, en la que cada vértice de G está en su propio conjunto.
    F = ∅;
    while (estaVacia(cp) == false)
      aristaVW = obtenerMin(cp);
      borrarMin(cp);
      int conjuntoV = hallar(conjuntos, aristaVW.de);
      int conjuntoW = hallar(conjuntos, aristaVW.a);
      if (conjuntoV ≠ conjuntoW)
        Añadir aristaVW a F;
        union(conjuntos, conjuntoV, conjuntoW);
    return;
```

8.4.2 Análisis

La cola de prioridad de aristas se puede implementar de manera eficiente con un montón, pues en este algoritmo no se usa la operación `decrementarClave`. El montón se puede construir en tiempo $\Theta(m)$. La eliminación de todas las aristas requiere un tiempo $\Theta(m \log m)$ en el peor caso, pero podría ser $\Theta(n \log m)$, que equivale a $\Theta(n \log n)$ si sólo es preciso procesar $O(n)$ aristas más ligeras para construir la colección de árboles abarcantes.

Como optimización adicional, si sabemos que el número de componentes conectados de G es `ncc`, sabremos que el número de aristas en la colección de árboles abarcantes es $n - \texttt{ncc}$, y el algoritmo podrá terminar tan pronto como haya añadido esa cantidad de aristas a F, que ahora es la colección de árboles abarcantes mínima, sin procesar las aristas restantes. La determinación del número de componentes conectados se puede efectuar en tiempo lineal. Esto haría posible aprovechar el caso favorable que mencionamos en el párrafo anterior.

En cuanto a las operaciones de Unión-Hallar, `hallar` se podría invocar aproximadamente $2m$ veces, mientras que `union` se invoca cuando más $n - 1$ veces. Así pues, el número total de operaciones Unión-Hallar efectuadas está acotado por $(2m + n)$. Supóngase $m \geq n$, que es lo normal, para simplificar las expresiones. Con la implementación de unión ponderada y compresión de caminos de la sección 6.6, el tiempo total de estas operaciones está en $O(m \lg^*(m))$, donde $\lg^*$ es la función de crecimiento muy lento de la definición 6.9.

Así pues, el tiempo de ejecución de peor caso del algoritmo MST de Kruskal está en $\Theta(m \log m)$. El algoritmo de Prim, algoritmo 8.1, está en $\Theta(n^2)$ en el peor caso. Cuál sea el mejor dependerá de los tamaños relativos de n y m. En el caso de grafos densos, el algoritmo de Prim es mejor. Con grafos ralos, el de Kruskal es más rápido que el algoritmo 8.1. No obstante, considérese la alternativa del ejercicio 8.9 y el hecho de que el algoritmo de Prim puede usar las estructuras de datos que vimos en la sección 8.2.8.

Si las aristas de G ya estuvieran ordenadas, se podría usar una cola de prioridad trivial y cada arista se podría borrar en tiempo $O(1)$, en cuyo caso el algoritmo de Kruskal se ejecutaría en tiempo $O(m \lg^*(m))$, que es muy bueno.

Ejercicios

Sección 8.2 Algoritmo de árbol abarcante mínimo de Prim

8.1 Dé un grafo no dirigido, ponderado, conectado y un vértice de inicio tales que ni el árbol de búsqueda primero en profundidad ni el árbol de búsqueda primero en amplitud sea un MST, sin importar cómo estén ordenadas las listas de adyacencia.

8.2 Sea T cualquier árbol abarcante de un grafo no dirigido G. Suponga que uv es cualquier arista de G que no está en T. Las demostraciones siguientes son fáciles si se usan las definiciones de árbol no dirigido, árbol abarcante y ciclo.

a. Sea G_1 el subgrafo que resulta de añadir uv a T. Demuestre que G_1 tiene un ciclo en el que participa uv, digamos $(w_1, w_2, \ldots, w_p, w_1)$, donde $p \geq 3$, $u = w_1$ y $v = w_p$.

b. Suponga que una arista cualquiera, $w_i w_{i+1}$ se elimina del ciclo creado en la parte (a), creando un subgrafo G_2 (que depende de i). Demuestre que G_2 es un árbol abarcante para G.

8.3 Suponga que T_1 y T_2 son árboles abarcantes mínimos distintos para el grafo G. Sea uv la arista más ligera que está en T_2 pero no está en T_1. Sea *xy cualquier* arista que está en T_1 pero no está en T_2. Demuestre que $W(xy) \geq W(uv)$.

8.4 Demuestre que si todos los pesos de las aristas de un grafo conectado no dirigido son distintos, sólo existe un árbol abarcante mínimo.

8.5

a. Describa una familia de grafos conectados, ponderados, no dirigidos G_n, para $n \geq 1$, tal que G_n tiene n vértices y el tiempo de ejecución del algoritmo MST de Prim (algoritmo 8.1) para G_n es lineal en n.

b. Describa una familia de grafos conectados, ponderados, no dirigidos G_n tal que G_n tiene n vértices y el algoritmo MST de Prim no efectúa comparaciones de pesos cuando G_n es la entrada. Una comparación de un peso con ∞ (`cp.oo`) no cuenta para este fin (porque el procedimiento `hallarMin` puede verificar si `verticeMin` es -1 para evitar dicha comparación). (El algoritmo requerirá un tiempo por lo menos proporcional a n porque debe hallar un árbol abarcante mínimo.)

8.6 Ejecute el algoritmo de árbol abarcante mínimo de Prim manualmente con el grafo de la figura 8.4(a), mostrando cómo evolucionan las estructuras de datos. Indique claramente cuáles aristas pasan a formar parte del árbol abarcante mínimo y en qué orden lo hacen.

a. Inicie en el vértice G.

b. Inicie en el vértice H.

c. Inicie en el vértice I.

8.7 Sea $G = (V, E, W)$ donde $V = \{v_1, v_2, \ldots, v_n\}$, $E = \{v_1 v_i \mid i = 2, \ldots, n\}$, y para $i = 2, \ldots, n$, $W(v_1 v_i) = 1$. Con este G como entrada y v_1 como vértice inicial, ¿cuántas comparaciones de pe-

sos de aristas efectuará el algoritmo MST de Prim, en total, para hallar aristas candidatas mínimas? (Resolver este problema podría sugerirle que guardar información acerca del ordenamiento de los pesos de las aristas candidatas podría reducir el número de comparaciones. Los dos ejercicios siguientes sugieren que quizá no sea fácil.)

8.8

a. ¿Cuántas comparaciones de pesos de aristas efectuará el algoritmo MST, en total, si la entrada es un grafo no dirigido completo con n vértices y v_1 es la arista inicial?

b. Suponga que los vértices son $v_1, \ldots, v_n$, y $W(v_i v_j) = n + 1 - i$ para $1 \leq i < j \leq n$. ¿Cuántas de las aristas son candidatas en algún momento durante la ejecución del algoritmo?

★ **8.9** Considere el almacenamiento de aristas candidatas en un montón minimizante (un montón en el que cada nodo es más pequeño que sus hijos, véase la sección 4.8.1). En este ejercicio evaluaremos el algoritmo MST de Prim bajo este supuesto, para grafos en general y para ciertas clases restringidas de grafos. Usaremos $|V| = n$ y $|E| = m$.

a. Determine el orden asintótico del número de comparaciones de pesos de aristas que efectuará el algoritmo MST de Prim, con base en la ecuación (8.1) en el peor caso. No olvide considerar el trabajo necesario para la operación `decrementarClave` cuando una arista candidata es sustituida por otra.

b. Una *familia de grafos de grado acotado* es cualquier familia para la cual existe una constante k tal que ningún vértice de cualquier grafo de la familia tiene grado mayor que k. Determine el orden asintótico, en función de n, del número de comparaciones de pesos de aristas que efectuaría el algoritmo MST de Prim con una familia de grado acotado.

★ **c.** Un *grafo plano* es un grafo conectado que se puede dibujar en un plano sin que haya cruces de aristas. Para esta clase, el teorema de Euler dice que $|V| - |E| + |F| = 2$, donde $|F|$ es el número de caras (regiones rodeadas por aristas, más una región desde las aristas externas hasta el infinito) que se forman al dibujar el grafo. Por ejemplo, si el grafo es un triángulo simple, tiene dos caras: una adentro del triángulo y otra fuera. La cara exterior se extiende hasta el infinito en todas direcciones. Determine el orden asintótico, en función de n, del número de comparaciones de pesos de aristas que efectuaría el algoritmo MST de Prim con un grafo plano. *Sugerencia:* Observe que todas las aristas están en dos caras.

8.10 El uso de la cola de prioridad se puede simplificar si se introducen inicialmente todos los vértices con `pesoBorde` = "∞" y `situacion` = `deborde`. El valor "∞" sólo tiene que ser mayor que el costo de cualquier arista que esté en el grafo; no tiene que representar realmente "infinito". Entonces `decrementarClave` reducirá el costo de un vértice por debajo de "∞" cuando se halle la primera conexión del vértice y no se necesitará `insertar`. Muestre las modificaciones del algoritmo 8.1 y las operaciones de cola de prioridad que implementan esta estrategia. ¿Mejora el orden asintótico?

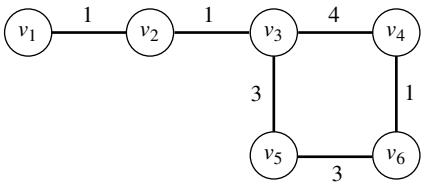


Figura 8.12 Grafo para el ejercicio 8.13

8.11 Supóngase que queremos usar el algoritmo de Prim con un grafo no dirigido ponderado del cual no se sabe si está conectado. Muestre cómo modificar el algoritmo de Prim para hallar una colección de árboles abarcantes mínima (definición 8.4) sin hallar primero los componentes conectados. Trate de no elevar el orden asintótico del algoritmo.

Sección 8.3 Caminos más cortos de origen único

8.12 Dé un grafo dirigido ponderado y un vértice de origen tal que ni el árbol de búsqueda primero en profundidad ni el árbol de búsqueda primero en amplitud sea un árbol de caminos más cortos, sin importar cómo estén ordenadas las listas de adyacencia.

8.13 Para el grafo de la figura 8.12, indique cuáles aristas estarían en el árbol abarcante mínimo construido por el algoritmo MST de Prim (algoritmo 8.1) y cuáles estarían en el árbol construido por el algoritmo de caminos más cortos de Dijkstra (algoritmo 8.2) empleando v_1 como origen.

8.14 ¿El algoritmo de caminos más cortos de Dijkstra (algoritmo 8.2) funciona correctamente si los pesos pueden ser negativos? Justifique su respuesta con un argumento o un contraejemplo.

8.15 He aquí las listas de adyacencia (con los pesos de las aristas entre paréntesis) para un grafo dirigido. Como ayuda, el grafo se muestra también en la figura 8.13.

A: *B*(4, 0), *F*(2.0)
B: *A*(1.0), *C*(3.0), *D*(4.0)
C: *A*(6.0), *B*(3.0), *D*(7.0)
D: *A*(6.0), *E*(2.0)
E: *D*(5.0)
F: *D*(2.0), *E*(3.0)

a. Este grafo dirigido tiene *tres* caminos más cortos de *C* a *E* (es decir, todos tienen el *mismo* peso total). Hállelos. (Enumere la sucesión de vértices de cada camino.)

b. ¿Cuál de estos caminos es el que hallaría el algoritmo de caminos más cortos de Dijkstra con $s = C$? (Dé una explicación convincente o muestre los pasos principales del algoritmo.)

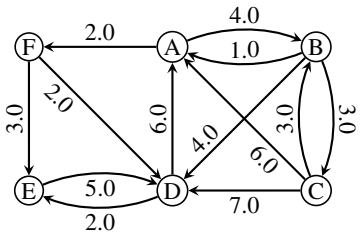


Figura 8.13 Digrafo para el ejercicio 8.15

c. Ejecute el algoritmo de caminos más cortos de Dijkstra manualmente con este grafo, mostrando cómo evolucionan las estructuras de datos, con $s = A$. Indique claramente cuáles aristas pasan a formar parte del árbol de caminos más cortos y en qué orden lo hacen.

d. Repita la parte (c) con $s = B$.

e. Repita la parte (c) con $s = F$.

8.16 Complete la demostración del teorema 8.7.

8.17 Explique cómo hallar un camino más corto real entre s y un vértice dado z utilizando el arreglo `padre` que el algoritmo de caminos más cortos de Dijkstra llena.

⋆ **8.18** Sea $G = (V, E)$ un grafo, y sean s y z vértices distintos. Como sugiere el ejercicio 8.15, puede haber más de un camino más corto entre s y z. Explique cómo modificaría el algoritmo de caminos más cortos de Dijkstra para determinar cuántos caminos más cortos distintos hay entre s y z.

8.19 Considere el problema de hallar sólo la distancia, pero no un camino más corto, desde s hasta un vértice dado z en un grafo ponderado. Bosqueje una versión modificada del algoritmo de caminos más cortos de Dijkstra para hacer esto tratando de eliminar la mayor cantidad de trabajo y de consumo de espacio extra posible. Indique cómo modificaría, si acaso, la estructura de datos empleada por el algoritmo, e indique qué trabajo o espacio eliminaría.

8.20 Algunos algoritmos para grafos se escriben bajo el supuesto de que la entrada siempre es un grafo completo (en el que una arista tiene peso ∞ o 0 para indicar su ausencia del grafo para el cual el usuario realmente quiere resolver el problema). Tales algoritmos suelen ser más cortos y "aseados" porque hay menos casos que considerar. En los algoritmos de las secciones 8.2 y 8.3, por ejemplo, no habría vértices no vistos porque todos los vértices estarían adyacentes a vértices del árbol construido hasta ese momento.

a. Con el objetivo de simplificar lo más posible, reescriba el algoritmo de caminos más cortos de Dijkstra bajo el supuesto de que $G = (V, E, W)$ es un grafo completo y que $W(uv)$ podría ser ∞. Describa cualesquier modificaciones que haría a las estructuras de datos empleadas.

b. Compare su algoritmo y estructuras de datos con las del texto, utilizando el criterio de sencillez, tiempo (peor caso y otros casos) y consumo de espacio (en el caso de grafos con muchas aristas de peso ∞ y grafos con pocas de esas aristas).

★ **8.21** Considere este enfoque general para calcular caminos más cortos desde el vértice s en un grafo ponderado $G = (V, E, W)$. Se forma G^T, el grafo transpuesto (véase la definición 7.10). Se define un arreglo llamado `llegar` con índices $1, \ldots, n$ y todos sus elementos inicializados con ∞. Se efectúa una búsqueda primero en profundidad completa de G^T y se calculan valores para `llegar[v]` según este esquema:

1. `llegar[s] = 0`;
2. Al retroceder de w a v, se calcula `llegar[w]` + $W(wv)$ y se compara con el valor previamente almacenado en `llegar[v]`. Si se halla un valor menor, se guarda como `llegar[v]`.

La intención es que `llegar[v]` sea la distancia de camino más corto entre s y v cuando termine la búsqueda primero en profundidad.

a. Complete el bosquejo anterior insertando enunciados en el esqueleto de búsqueda primero en profundidad para grafos dirigidos, algoritmo 7.3.

b. ¿El algoritmo halla caminos más cortos en todos los casos? Demuestre que lo hace o halle un contraejemplo.

c. ¿Con qué clase (muy conocida) de grafos el algoritmo halla caminos más cortos en todos los casos? Demuestre su respuesta. *Sugerencia:* ¿Qué restricción del grafo permitiría efectuar con éxito la demostración del inciso (b)?

Sección 8.4 Algoritmo de árbol abarcante mínimo de Kruskal

8.22 Demuestre el lema 8.8.

8.23 Demuestre el teorema 8.9. *Sugerencia:* Use la propiedad MST (definición 8.2).

8.24 Halle el árbol abarcante mínimo que el algoritmo de Kruskal (algoritmo 8.3) produciría para el grafo de la figura 8.14, suponiendo que las aristas están ordenadas como se muestra.

Problemas adicionales

8.25 En este ejercicio el lector desarrollará un esqueleto de *búsqueda de primero el mejor*, análogo al esqueleto de búsqueda primero en amplitud del capítulo 7.

a. Considere si usará la estrategia del ejercicio 8.20 (nada de inserciones, crear la cola de prioridad con todos los elementos presentes y pesos infinitos en caso necesario) o el TDA `CPMin` dado en las figuras 8.6 a 8.8 como base para su esqueleto. ¿Cuál ofrece más generalidad? ¿Cuál es probable que sea más eficiente? ¿Son más o menos iguales los dos?

b. Escriba el esqueleto con la estrategia elegida.

c. Muestre cómo modificar su esqueleto (insertando unos cuantos enunciados en ciertos puntos) para producir el algoritmo de Prim, y luego para producir el algoritmo de Dijkstra.

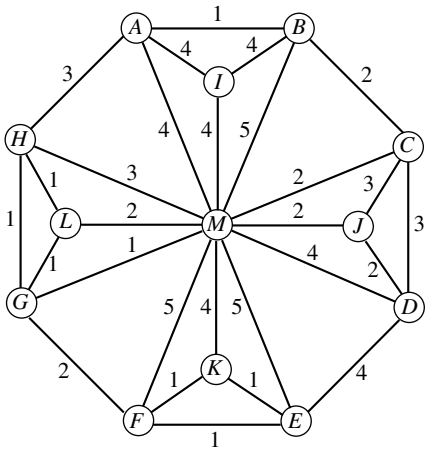


Figura 8.14 Aristas ordenadas: *AB, EF, EK, FK, GH, GL, GM, HL, BC, CM, DJ, FG, JM, LM, AH, CD, CJ, HM, AI, AM, BI, DE, DM, IM, KM, BM, EM, FM.*

d. Comente la interfaz de bosques de apareamiento con su esqueleto. ¿La interfaz es independiente de las aplicaciones con las que podría usarse su esqueleto o tendría que modificarse dependiendo de la aplicación?

8.26 Suponga que quiere hallar un camino más corto de s a w en un grafo G en el que la longitud de un camino es simplemente el número de aristas del camino (por ejemplo, para planear un viaje en avión con el mínimo de escalas). ¿Cuál de los algoritmos o estrategias de recorrido de este capítulo o del capítulo 7 *podría* usar? ¿Cuál *usaría*, y por qué?

8.27 Suponga que necesita determinar si un grafo grande está conectado o no. El grafo tiene n vértices, $V = \{1, \ldots, n\}$ y m aristas, donde m es mucho mayor que n. La entrada consistirá en los enteros n y m y una sucesión de aristas (pares de vértices). No hay suficiente espacio para almacenar todo el grafo; se dispone de cn unidades de espacio, donde c es una constante pequeña, pero no se puede usar espacio proporcional a m. Así pues, es posible procesar cada arista cuando se lee, pero no guardar las aristas ni volverlas a leer. Describa un algoritmo para resolver el problema. ¿Cuánto tiempo tarda su algoritmo en el peor caso?

Programas

Cada uno de los siguientes programas de tarea requiere un procedimiento que lee una descripción de un grafo con aristas ponderadas y establece las listas de adyacencia. Basta una modificación menor del procedimiento de carga de grafos del capítulo 7. Suponga que la entrada contiene el número de vértices seguido de una sucesión de líneas, cada una de las cuales contiene un par de no-

dos que representan una arista y un tercer número que representa su peso. Escriba este procedimiento de modo que, con cambios pequeños, se pueda usar con cualquiera de los problemas.

Se deben escoger datos de prueba que permitan probar todos los aspectos del programa. Incluya algunos de los ejemplos del texto.

1. Algoritmo de árbol abarcante mínimo de Prim, algoritmo 8.1. El programa debe completar el algoritmo, construyendo una estructura de datos para registrar el árbol abarcante mínimo hallado. La salida debe provenir de un procedimiento aparte y debe incluir el grafo, el conjunto de aristas del árbol, junto con sus pesos, y el peso total del árbol.
2. Algoritmo de árbol abarcante mínimo de Kruskal, algoritmo 8.3. El programa debe completar el algoritmo, construyendo una estructura de datos para registrar el árbol abarcante mínimo hallado. La salida debe provenir de un procedimiento aparte y debe incluir el grafo, el conjunto de aristas del árbol, sus pesos y el peso total del árbol.
3. Algoritmo de camino más corto de Dijkstra, algoritmo 8.2. El programa debe completar el algoritmo, construyendo una estructura de datos para registrar los caminos más cortos hallados. La salida debe provenir de un procedimiento aparte y debe incluir el grafo (o digrafo), el vértice de origen, cada uno de los vértices a los que se puede llegar desde el origen, junto con las aristas del camino más corto hallado a ese vértice, sus pesos y el peso total del camino.
4. Después de escribir el programa 1 o el programa 3, modifíquelo para implementar la cola de prioridad con bosques de apareamiento. Efectúe pruebas de tiempo con algunos grafos grandes y compare los tiempos antes y después de las modificaciones.

Notas y referencias

El primer algoritmo de árbol abarcante mínimo se debe a Prim (1957). El algoritmo de camino más corto de origen único se debe a Dijkstra (1959), pero ese artículo no trata la implementación. Dijkstra (1959) también describe un algoritmo de árbol abarcante mínimo parecido al de Prim. La terminología para clasificar los vértices en las secciones 8.2 y 8.3 (por ejemplo, *vértice de borde*) se tomó de Sedgewick (1988). En Notas y referencias del capítulo 6 se mencionan alternativas para implementar colas de prioridad, como bosques de apareamiento, montones de apareamiento y montones de Fibonacci. La cota superior para los algoritmos de Prim y Dijkstra, utilizando montones de apareamiento, con grafos para los que $m = \Theta(n^{1+c})$, se tomó de Fredman (1999).

En algunas aplicaciones es necesario hallar un árbol abarcante con peso mínimo entre los que satisfacen otros criterios requeridos por el problema, así que es útil tener un algoritmo que genere árboles abarcantes en orden según su peso para poder determinar si cada uno satisface los otros criterios. Gabow (1977) presenta algoritmos que hacen esto.

La estrategia de Kruskal para hallar árboles abarcantes mínimos se tomó de Kruskal (1956). La implementación empleando programas de equivalencia al parecer era folklore; se menciona en Hopcroft y Ullman (1973), quienes informan que M. D. McIlroy y R. Morris llevaron a cabo tal implementación. Gran parte del material de esta sección, junto con aplicaciones adicionales y extensiones, aparece en Aho, Hopcroft y Ullman (1974). El lector puede hallar otros algoritmos de caminos más cortos, incluidos algunos que no requieren pesos de arista no negativos, en Cormen, Leiserson y Rivest (1990).

En la sección 7.2 presentamos varias preguntas que podrían hacerse acerca de los grafos y digrafos. Una que no contestamos en este libro es: ¿Qué cantidad de un producto puede fluir de un vértice a otro dadas capacidades de las aristas? Se trata del problema de flujo por red; tiene muy diversas soluciones y aplicaciones. Los lectores interesados pueden consultar Even (1979), Ford y Fulkerson (1962), Tarjan (1983), Wilf (1986) y Cormen, Leiserson y Rivest (1990).

9

Cierre transitivo, caminos más cortos de todos los pares

9.1 Introducción

En este capítulo estudiaremos dos problemas relacionados que se pueden describir informalmente como preguntas acerca de *todos* los pares de vértices de un grafo:

1. ¿Existe un camino de u a v?
2. ¿Cuál es el camino *más corto* de u a v?

En los capítulos 7 y 8 vimos algoritmos para estos problemas en los casos en que el primer vértice es especial y sólo el segundo vértice puede ser cualquier otro vértice del grafo. En este capítulo estudiaremos el problema más global.

La principal idea algorítmica que presentamos en este capítulo tiene una aplicación muy amplia; fue descubierta de manera independiente, para diferentes aplicaciones, por Kleene (para la síntesis de un lenguaje regular, tema que no se cubre en este libro), por Warshall (para el cierre transitivo) y por Floyd (para los caminos más cortos de todos los pares). Por consiguiente, se le conoce como algoritmo de Kleene-Floyd-Warshall, el cual es aplicable a toda una clase de problemas llamados problemas de *cierre semianular* que rebasan el alcance de esta obra. En Notas y referencias al final del capítulo se sugieren lecturas adicionales.

9.2 Cierre transitivo de una relación binaria

En esta sección definiremos el *cierre transitivo* en términos de relaciones binarias y examinaremos su relación con los caminos en grafos dirigidos. También introduciremos cierta notación que se usa en todo el capítulo. Luego examinaremos unas cuantas estrategias sencillas para calcular el cierre transitivo. En secciones posteriores presentaremos algoritmos más avanzados.

9.2.1 Definiciones y antecedentes

Sea S un conjunto con elementos $s_1, s_2, \ldots$. Recordemos (sección 1.3.1) que una *relación binaria* sobre S es un subconjunto, digamos A, de $S \times S$. Si $(s_i, s_j) \in A$, decimos que s_i está relacionado por A con s_j y usaremos la notación $s_i A s_j$.

Supóngase que S tiene n elementos. La relación A puede representarse mediante una matriz booleana $n \times n$ con los elementos

$$a_{ij} = \begin{cases} \textit{verdadero} & \text{si } s_i A s_j \\ \textit{falso} & \text{en los demás casos.} \end{cases}$$

Comenzaremos con esta representación, pero más adelante consideraremos representaciones que usan los bits 1 y 0 para indicar *verdadero* y *falso*; también, en los diagramas usaremos 1 y 0. Hablando de matrices booleanas, el término *matriz cero*, denotada por 0, se refiere a la matriz en la que todos los elementos son *falso*, y la *matriz de identidad*, denotada por I, es la matriz en la que todos los elementos son *falso* salvo los que están en la diagonal principal (a_{ii}), que son *verdadero*.

La relación de adyacencia sobre el conjunto de vértices de un grafo, que usamos ampliamente en el capítulo 7, es un ejemplo importante de relación. Otros ejemplos comunes son las relaciones de equivalencia y los órdenes parciales. Por otra parte, cualquier relación binaria A sobre un conjunto S se puede interpretar como el grafo dirigido

$$G = (S, A); \tag{9.1}$$

es decir, los elementos de S se interpretan como los vértices, en tanto que los pares ordenados en A se interpretan como las aristas.

Usaremos la misma letra (mayúscula) para denotar una relación y su representación matricial (que supone un orden específico de los elementos del conjunto subyacente), las letras minúsculas corresponden a los elementos de la matriz. A menos que se diga otra cosa, supondremos que el conjunto en cuestión es $S = \{s_1, ..., s_n\}$.

Notación de operadores booleanos

En el pseudocódigo usaremos los símbolos matemáticos "$\wedge$", "$\vee$" y "$\neg$" para denotar los operadores lógicos *y* (*and*), *o* (*or*) y *no* (*not*), respectivamente. Algunos autores llaman a "$\vee$" la *suma booleana*, siguiendo la costumbre en ingeniería eléctrica de usar los símbolos "+" para el *o* binario y Σ para el *o* multivías; emplearemos esta notación en ciertos casos, pero no debe confundirse con el operador *o exclusivo* (que también se denota a veces con "+"); en este capítulo no usaremos el *o exclusivo*. Hablando de matrices booleanas, $A \vee B$ significa que cada elemento se calcula como $(a_{ij} \vee b_{ij})$. (Las definiciones de los demás operadores lógicos son similares, pero no tendremos oportunidad de usarlos.)

Cierre transitivo

Recordemos la definición 1.2, según la cual una relación A sobre S es *transitiva* si y sólo si, para todas s_i, s_j y s_k en S: $s_i A s_j$ y $s_j A s_k$ implica $s_i A s_k$. Las relaciones de equivalencia y los órdenes parciales son relaciones transitivas. Por lo regular, la relación de adyacencia de un grafo no es transitiva.

Definición 9.1 Cierre transitivo

Sea S un conjunto y sea A una relación binaria sobre S. Sea $G = (S, A)$, como en la ecuación (9.1). El *cierre transitivo reflexivo* de A (llamado *cierre transitivo* de A para abreviar) es la relación binaria R definida por: $s_i R s_j$ si y sólo si existe en G un camino de s_i a s_j. El cierre transitivo de la relación de adyacencia de un grafo también se denomina *relación de alcanzabilidad* o *asequibilidad*.

Cabe señalar que el cierre transitivo (reflexivo) de A es reflexivo porque existe un camino de longitud cero de cada vértice a sí mismo. El *cierre transitivo no reflexivo* de A se define de forma similar, con la salvedad de que el camino de s_i a s_j no puede estar vacío. ■

El cierre transitivo de una relación transitiva y reflexiva A es la relación A misma. En términos más generales, puede demostrarse que el cierre transitivo de cualquier relación A es la relación *mínima* R tal que $A \subseteq R$ y R es transitiva y reflexiva.

Ejemplo 9.1 Cierre transitivo de una relación

Para la relación A que sigue, el cierre transitivo es R.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \qquad R = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Podemos verificar que R es transitiva por inspección. Por ejemplo, s_1Rs_5 y s_5Rs_3, por lo que deberá ser cierto que s_1Rs_3, y así es. ■

En las secciones 9.2, 9.3, 9.5 y 9.6 estudiaremos diversos métodos para hallar el cierre transitivo de una relación. La aplicación a grafos es útil. La forma en que se dan las entradas depende de la manera en que se presenta el problema en una aplicación dada. En todos los casos supondremos que $|S| = n$ y $|A| = m$.

9.2.2 Determinación de la matriz de alcanzabilidad mediante búsqueda primero en profundidad

Una forma obvia de construir R, la matriz de alcanzabilidad de un grafo dirigido $G = (S, A)$, consiste en efectuar una búsqueda primero en profundidad (véase la sección 7.3) desde cada vértice para hallar todos los vértices a los que se pueda llegar desde él. En un principio, R sería la matriz cero. La acción de visitar, o procesar, un vértice s_j encontrado en la búsqueda primero en profundidad desde s_i consistiría en asignar *verdadero* a r_{ij}. Así, cada búsqueda primero en profundidad llena una fila de R. Esto podría parecer ineficiente y poco astuto porque durante una búsqueda primero en profundidad desde, digamos s_i, podrían colocarse elementos en filas distintas de la i-ésima; específicamente, cuando se encuentra un vértice s_j, podría asignarse *verdadero* a r_{kj} para toda k tal que s_k está en el camino de s_i a s_j. Estos vértices s_k son grises y pueden hallarse en la pila. ¿Qué tan importante es esta modificación? ¿Hace innecesario realizar una búsqueda primero en profundidad desde s_k? ¿Cómo afecta la cantidad de trabajo que se efectúa en el peor caso?

Puesto que ya estudiamos la búsqueda primero en profundidad con muchos ejemplos en el capítulo 7, no deduciremos los detalles de un algoritmo en esta seción, sólo limitaremos a hacer unos cuantos comentarios acerca de la cantidad de trabajo efectuado. Si usamos la estructura de listas de adyacencia descrita en el capítulo 7 para representar G y efectuamos una búsqueda primero en profundidad para cada vértice, el tiempo de ejecución de peor caso estará en $\Theta(nm)$. La inserción de valores *verdadero* en más de una fila de R durante cada búsqueda primero en profundidad, como acabamos de sugerir, puede mejorar el comportamiento del algoritmo con muchos grafos, pero el peor caso seguirá estando en $\Theta(nm)$. (Véase el ejercicio 9.2.)

En el capítulo 7 definimos la condensación de un grafo dirigido. Informalmente, la condensación es el digrafo que se obtiene al reducir cada componente fuertemente conectado a un solo punto; es acíclico. Mencionamos que algunos problemas podrían simplificarse si trabajamos con la condensación en lugar del digrafo original. La relación de alcanzabilidad para un digrafo $G = (S, A)$ se puede calcular como sigue:

1. Determinar los componentes fuertes de G (en tiempo $\Theta(n + m)$). Sea $G\downarrow$ la condensación de G.
2. Determinar la relación de alcanzabilidad para $G\downarrow$. (Se puede usar cualquiera de los métodos que presentamos en este capítulo.)
3. Expandir la relación de alcanzabilidad para $G\downarrow$ sustituyendo cada vértice de $G\downarrow$ por todos los vértices de G que se redujeron a él (en tiempo $O(n^2)$).

La cantidad de trabajo efectuada en el paso 2, y por ende con este método en general, depende del digrafo específico de que se trate. Si G tiene varios componentes fuertes grandes, la reducción a $G\downarrow$ podría ahorrar mucho tiempo.

Una búsqueda primero en profundidad eficiente emplea listas de adyacencia. En la sección que sigue presentaremos un algoritmo $\Theta(n^3)$ relativamente sencillo para hallar la matriz de alcanzabilidad empleando la matriz de adyacencia como representación del digrafo.

9.2.3 Cierre transitivo por atajos

Si interpretamos una relación binaria A sobre un conjunto finito S como un grafo dirigido, hallar los elementos de R, el cierre transitivo de la relación, corresponderá a insertar aristas en el digrafo. En particular, para cada par de aristas $s_i s_k$ y $s_k s_j$ insertadas hasta el momento, añadimos la arista $s_i s_j$. Es decir, podemos concluir que $s_i R s_j$ si ya sabemos que, para alguna k, $s_i R s_k$ y $s_k R s_j$. Podemos ver a $s_i R s_j$ como un "atajo" en el digrafo correspondiente que nos permite ir de s_i a s_j con un solo paso en lugar de dos. La relación R es transitiva si no es posible añadir más atajos. Así pues, no es difícil convencernos de que el algoritmo siguiente calcula R.

Algoritmo 9.1 Cierre transitivo por atajos

Entradas: A y n, donde A es una matriz **boolean** de $n \times n$ que representa una relación binaria.

Salidas: R, la matriz **boolean** para el cierre transitivo de A.

```
void cierreTransitivoSimple(boolean[][] A, int n, boolean[][] R)
    int i, j, k;
    Copiar A en R.
    Asignar true a todos los elementos de la diagonal principal, r_ii.
    while (cualquier elemento de R cambie durante una pasada completa)
      for (i = 1; i <= n; i ++)
        for (j = 1; j <= n; j ++)
          for (k = 1; k <= n; k ++)
            r_ij = r_ij ∨ (r_ik ∧ r_kj);
```

Ejemplo 9.2 Cierre transitivo por atajos

Consideremos la relación A del ejemplo 9.1. El digrafo correspondiente se muestra en la figura 9.1(a). El algoritmo 9.1 añade "autoaristas" antes de que inicie su ciclo **while**, aunque no se muestran en la figura. Después de una pasada por el ciclo **while** del algoritmo 9.1, se han añadido las aristas que se muestran como líneas punteadas en la figura 9.1(b). Obsérvese que se pudo añadir (5, 2) aunque el camino de 5 a 2 tiene longitud 3, porque (4, 2) se había añadido antes. En cambio, (1, 3) no puede añadirse en esta pasada. Durante la segunda pasada se añade (1, 3), como se muestra en la figura 9.1(c). En la tercera pasada no se añaden aristas. ■

La figura 9.1 ilustra que no es posible omitir el ciclo **while**. Cuando se consideran por primera vez un s_i y un s_j específicos, es posible que no haya un s_k que los una. Más adelante en el procesamiento, gracias a la inserción de otras aristas, podría ser posible insertar $s_i s_j$; por tanto, debemos reconsiderarlo. La complejidad del algoritmo 9.1 es proporcional a n^3 veces el número de repeticiones del ciclo **for** triple. La investigación de esta cifra se deja para el ejercicio 9.4, puesto que en la sección 9.3 modificaremos el algoritmo para reducir la cantidad de trabajo efectuada.

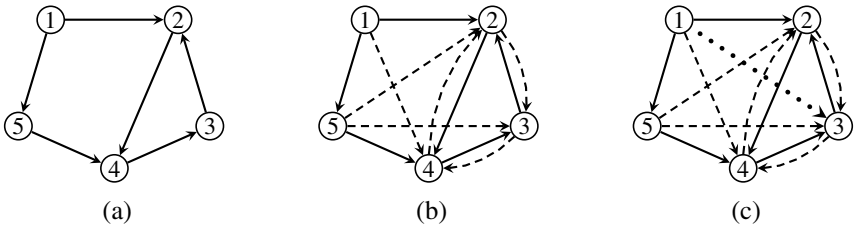


Figura 9.1 (a) Grafo dirigido que representa una relación *A*. (b) Las líneas de guiones muestran los atajos añadidos después de una pasada. (c) La línea de puntos muestra la adición de otro atajo en la segunda pasada. Se han omitido las autoaristas.

Posteriormente, en la sección 9.5, volveremos a la idea del algoritmo 9.1 y la reinterpretaremos en términos de la multiplicación de matrices booleanas.

Llamamos al trabajo efectuado en el enunciado "$r_{ij} = r_{ij} \vee (r_{ik} \wedge r_{kj})$" *procesamiento de la tripleta* (*i, k, j*). En la figura 9.1, si las tripletas se procesaran en orden inverso, de modo que (5, 4, 3) se procesara pronto, ninguna tripleta tendría que considerarse dos veces. ¿Existe algún orden que siempre elimine la necesidad de procesar cualquier tripleta más de una vez? ¿O, sea cual sea el orden que probemos, siempre podremos hallar un ejemplo con el que se requiera repetición? Sugerimos al lector tratar de contestar estas preguntas antes de continuar.

9.3 Algoritmo de Warshall para cierre transitivo

El algoritmo de Warshall es simplemente un algoritmo que procesa las tripletas mencionadas en la sección 9.2.3 en el orden correcto: específicamente, variando *k* en el ciclo más exterior. Primero describiremos el algoritmo básico empleando matrices booleanas. Después presentaremos una demostración de corrección, y por último describiremos una optimación que emplea cadenas de bits.

9.3.1 El algoritmo básico

Algoritmo 9.2 Cierre transitivo (de Warshall)

Entradas: *A* y *n*, donde *A* es una matriz $n \times n$ que representa una relación binaria.

Salidas: *R*, la matriz $n \times n$ para el cierre transitivo de *A*.

```
void cierreTransitivo(boolean[][] A, int n, boolean[][] R)
    int i, j, k;
    Copiar A en R.
    Asignar true a todos los elementos de la diagonal principal, r_ii.
    for (k = 1; k <= n; k ++)
      for (i = 1; i <= n; i ++)
        for (j = 1; j <= n; j ++)
          r_ij = r_ij ∨ (r_ik ∧ r_kj);
```

Es evidente que el número total de tripletas procesadas está en n^3. La inicialización de R toma un tiempo $\Theta(n^2)$, así que el número de elementos de matriz examinados y/o modificados con cualquier entrada está en $\Theta(n^3)$.

La corrección del algoritmo se basa en la definición que sigue y del lema que se presenta después.

Definición 9.2 Vértice intermedio de número más alto

Sea G un grafo dirigido cuyos vértices están indizados por los enteros 1, 2, ..., n y que se denotan con $(s_1, s_2, \ldots, s_n)$; es decir, se les considera una sucesión ordenada, no sólo un conjunto. Para cualquier camino no vacío en G, el *vértice intermedio de número más alto* de ese camino es un vértice que no es el inicial ni el final y tiene el índice más alto de todos los vértices intermedios del camino. Si el camino consta de una sola arista, se considera que el vértice intermedio de número más alto es 0. ■

Lema 9.1 En el algoritmo 9.2, sea $r_{ij}^{(0)}$ el valor de r_{ij} después de las inicializaciones y, para cada k en 1, ..., n, sea $r_{ij}^{(k)}$ el valor de r_{ij} después de la k-ésima ejecución del cuerpo del ciclo "**for** (k ...)". Si existe cualquier camino simple de s_i a s_j ($i \neq j$) cuyo vértice intermedio de número más alto es s_k, entonces $r_{ij}^{(k)} = verdadero$.

Demostración La demostración es por inducción con k, el número de veces que se ha ejecutado el ciclo "**for** (k ...)". El caso base es $k = 0$, es decir, cuando todavía no se ha ejecutado el ciclo **for** pero ya se terminó la inicialización. En tal caso $r_{ij}^{(0)} = a_{ij}$, así que $r_{ij}^{(0)} = verdadero$ si y sólo si existe un vértice $s_i s_j$. En este caso el vértice intermedio de número más alto es 0.

Para $k > 0$, suponemos que el lema se cumple para $0 \leq h < k$. El camino simple, llamémoslo P_{ij}, de s_i a s_j con vértice intermedio de número más alto s_k se puede dividir en dos caminos no vacíos, P_{ik} de s_i a s_k, y P_{kj} de s_k a s_j, como se muestra en la figura 9.2.

Los vértices intermedios de número más alto en P_{ik} y P_{kj} tienen índices estrictamente menores que k, porque P_{ij} es un camino simple. Por la hipótesis inductiva, $r_{ik}^{(k)} = verdadero$ para algún $h < k$. Sin embargo, una vez que r_{ik} se vuelve *verdadero* gracias al operador "$\vee$", seguirá siendo

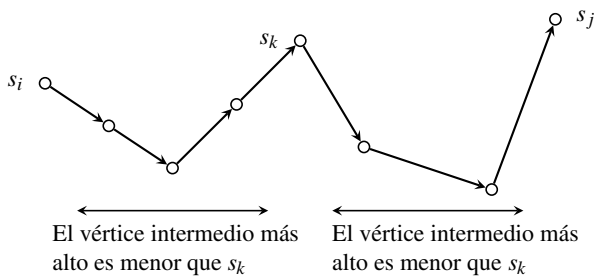


Figura 9.2 Camino de s_i a s_j con vértice intermedio de número más alto s_k. (Las posiciones verticales de los vértices reflejan su número de vértice.)

verdadero, así que $r_{ik}^{(k-1)} = verdadero$. Vale un argumento similar para $r_{kj}^{(k-1)}$. Por tanto, la ejecución del enunciado "$r_{ij} = r_{ij} \vee (r_{ik} \wedge r_{kj})$" hace que $r_{ij}^{(k)}$ sea *verdadero*. □

La clave para la demostración del algoritmo de Warshall es la forma del camino de la figura 9.2, que es fácil de recordar porque semeja una W en este ejemplo. La parte importante de esta forma es el pico de enmedio; los demás aspectos podrían variar considerablemente.

Teorema 9.2 Cuando el algoritmo 9.2 termina, R es la matriz que representa el cierre transitivo de A.

Demostración Obsérvese que existe un camino de s_i a s_j si y sólo si existe un camino *simple* de s_i a s_j. Por inicialización, todo r_{ii} es *verdadero*. Por el lema 9.1, r_{ij} es *verdadero* para todos los pares tales que hay un camino simple no vacío de s_i a s_j (puesto que el valor final de r_{ij} es $r_{ij}^{(n)}$, y una vez que $r_{ij}^{(k)}$ es *verdadero* ello asegura que $r_{ij}^{(n)}$ sea *verdadero*). Para cualquier $s_i \neq s_j$, dado que se asigna a r_{ij} el valor inicial *falso* a menos que exista una arista (s_i, s_j), y sólo se puede asignar *verdadero* a r_{ij} dentro del ciclo cuando se halla un camino simple, se sigue que r_{ij} es *falso* si *no* existe un camino de s_i a s_j. □

9.3.2 Algoritmo de Warshall para matrices de bits

Si las matrices A y R se almacenan con un elemento por bit, el algoritmo de Warshall tiene la implementación rápida siguiente que emplea la instrucción *or* por bits (o *suma* booleana, o unión) con la que cuenta la mayor parte de las computadoras de uso general. En Java, C y C++, el *or* por bits con enteros se implementa mediante el operador "|". En nuestro pseudocódigo seguiremos usando "$\vee$".

Definición 9.3 Cadena de bits, matriz de bits, `ORporbits`

Una *cadena de bits* de longitud n es una sucesión de n bits que ocupan posiciones de memoria continuas a partir de una frontera de palabra de computadora, rellenándose hasta la próxima frontera de palabra al final, si es necesario. Es decir, si una palabra de computadora abarca c bits, una cadena de n bits se almacenará en un arreglo de $\lceil n/c \rceil$ palabras de computadora.

Una *matriz de bits* es un arreglo de cadenas de bits, cada una de las cuales representa una fila de la matriz. Si A es una matriz de bits, $A[i]$ denota la i-ésima fila de A y es una cadena de bits. También, a_{ij} denota el j-ésimo bit de $A[i]$.

Definimos el procedimiento `ORporbits(a, b, n)`, donde a y b son cadenas de bits y n es un entero, para calcular $a \vee b$ bit por bit para n bits, dejando el resultado en a. ■

Algoritmo 9.3 Cierre transitivo para matrices de bits (de Warshall)

Entradas: A y n como en el algoritmo 9.2, pero A es una matriz de bits. (En el pseudocódigo suponemos que ya se definió la clase `MatrizDeBits`.)

Salidas: R, el cierre transitivo de A, que también es una matriz de bits.

```
void matricesBitsWarshall(MatrizDeBits A, int n, MatrizDeBits R)
    int i, k;
    Copiar A en R.
    Asignar 1 a toda r_ii.
    for (k = 1; k <= n; k ++)
      for (i = 1; i <= n; i ++)
        if (r_ik == 1)
          ORporbits(R[i], R[k], n);
```

Se efectúan cuando más n^2 operaciones *or* por bits con filas de R. Sin embargo, es posible que una fila no quepa en una sola palabra de memoria, en cuyo caso se necesitaría más de una instrucción *or* para implementar `ORporbits`. (En algunas computadoras una instrucción de máquina calcula el *or* booleano de dos cadenas de bits largas —digamos de hasta 256 bytes, es decir, 2048 bits— aunque el tiempo que toma ejecutar la instrucción depende de la longitud de los operandos.) El número de operaciones *or* necesarias para cada fila es $\lceil n/c \rceil$, donde c es el tamaño de palabra (o el tamaño del operando de la instrucción *or* booleana), así que el algoritmo 9.3 ejecuta $\lceil n^3/c \rceil$ instrucciones *or* booleanas en el peor caso. La complejidad está en $\Theta(n^3)$, pero el múltiplo constante de n^3 es pequeño.

9.4 Caminos más cortos de todos los pares en grafos

En el capítulo 8 estudiamos el algoritmo de Dijkstra (algoritmo 8.2), que halla un camino más corto y la distancia entre un *vértice de origen* dado y todos los demás vértices de un grafo ponderado. El algoritmo utiliza la estructura de listas de adyacencia y se ejecuta en tiempo $\Theta(n^2)$ en el peor caso. (Los pesos no pueden ser negativos.) Ahora consideraremos el problema siguiente:

Problema 9.1 Caminos más cortos de todos los pares

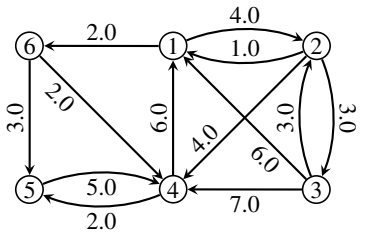
Dado un grafo ponderado $G = (V, E, W)$ con $V = \{v_1, \ldots, v_n\}$, representado por la matriz de pesos con elementos

$$w_{ij} = \begin{cases} W(v_i v_j) & \text{si } v_i v_j \in E, \\ \infty & \text{si } v_i v_j \notin E \text{ e } i \neq j, \\ \text{mín } (0, W(v_i v_j)) & \text{si } v_i v_i \in E, \\ 0 & \text{si } v_i v_i \notin E, \end{cases} \tag{9.2}$$

calcular la matriz $n \times n$ D definida por d_{ij} = la distancia de camino más corto de v_i a v_j. (La *distancia* es el peso de un camino de peso mínimo.) ■

En la figura 9.3 se da un ejemplo. El problema podría extenderse pidiendo una *tabla de rutas* de la cual pueden extraerse caminos más cortos. Si existen ciclos con peso negativo, no estará definido un camino más corto para algunos pares de vértices; los caminos pueden hacerse más cortos dando vueltas por este ciclo un número arbitrario de veces.

Una estrategia para calcular D (si G no tiene pesos negativos) sería usar el algoritmo 8.2 repetidamente, comenzando en cada ocasión en otro vértice hasta terminarlos. Sin embargo, pode-



$$W = \begin{bmatrix} 0 & 4 & \infty & \infty & \infty & 2 \\ 1 & 0 & 3 & 4 & \infty & \infty \\ 6 & 3 & 0 & 7 & \infty & \infty \\ 6 & \infty & \infty & 0 & 2 & \infty \\ \infty & \infty & \infty & 5 & 0 & \infty \\ \infty & \infty & \infty & 2 & 3 & 0 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 4 & 7 & 4 & 5 & 2 \\ 1 & 0 & 3 & 4 & 6 & 3 \\ 4 & 3 & 0 & 7 & 9 & 6 \\ 6 & 10 & 13 & 0 & 2 & 8 \\ 11 & 15 & 18 & 5 & 0 & 13 \\ 8 & 12 & 15 & 2 & 3 & 0 \end{bmatrix}$$

Figura 9.3 La matriz de pesos y la de distancia para un digrafo

mos usar una extensión del algoritmo de Warshall, propuesta por R. W. Floyd, para obtener un algoritmo más ágil (eliminando las estructuras de datos empleadas en el algoritmo 8.2).

¿Cómo calculamos $D[i][j]$? Un camino más corto podría pasar por cualquiera de los demás vértices en cualquier orden. Al igual que en el algoritmo de Warshall, clasificamos los caminos según su vértice intermedio de número más grande (véase la definición 9.2).

Recordemos la propiedad de camino más corto del lema 8.5: si un camino más corto de v_i a v_j pasa por un vértice intermedio v_k, los segmentos de ese camino, de v_i a v_k y de v_k a v_j, son en sí caminos más cortos. Si escogemos k de modo que tenga el índice más grande de todos los vértices intermedios del camino de v_i a v_j (suponiendo que el camino tenga más de una arista), cada uno de los segmentos mencionados tendrá un vértice intermedio de número más alto cuyo índice será estrictamente menor que k. (Véase la figura 9.2, que muestra la misma idea para el algoritmo de Warshall.) Esto sugiere calcular una matriz de distancia D en rondas, según la ecuación de recurrencia siguiente.

$$D^{(0)}[i][j] = w_{ij}$$
$$D^{(k)}[i][j] = \text{mín}\,(D^{(k-1)}\,[i][j],\, D^{(k-1)}\,[i][k] + D^{(k-1)}\,[k][j]) \qquad (9.3)$$

donde w_{ij} se definió en la ecuación (9.2). Por las observaciones anteriores acerca de la propiedad de camino más corto y el mismo argumento que se usó en el lema 9.1, se puede demostrar el lema siguiente (la demostración se deja como ejercicio).

Lema 9.3 Para cada k en 0, …, n, sea $d_{ij}^{(k)}$ del peso de un camino simple más corto de v_i a v_j con vértice intermedio de número más alto v_k, y defínase $D^{(k)}[i][j]$ según la ecuación (9.3). Entonces, $D^{(k)}[i][j] \leq d_{ij}^{(k)}$. □

Ejemplo 9.3 Cálculo de una matriz de distancias

El cálculo de $D^{(6)}[4][3]$ para el digrafo de la figura 9.3 ilustra el caso en que $D^{(6)}[4][3] < d_{43}^{(6)}$. $D^{(5)}[4][3] = 13$ (porque el mejor camino de 4 a 3 empleando sólo {1, …, 5} es el camino 4, 1, 2, 3, que pesa 13). Permitir el uso del vértice 6 no da un mejor camino. Tenemos $D^{(5)}[4][6] = 8$ (por el camino 4, 1, 6) y $D^{(5)}[6][3] = 15$ (por el camino 6, 4, 1, 2, 3), así que $d_{43}^{(6)} = 23$.

El cálculo de $D^{(6)}[1][5]$ ilustra un caso en el que el vértice 6 sí ayuda. $D^{(5)}[1][5] = 10$ (porque el mejor camino de 1 a 5 empleando sólo $\{1, \ldots, 5\}$ es el camino 1, 2, 4, 5, que pesa 10). Permitir el uso del vértice 6 da un camino más corto: 1, 6, 5, que pesa 5. Obtenemos esto sumando $D^{(5)}[1][6] = 2$ y $D^{(5)}[6][5] = 3$. ■

La ecuación (9.3) calcula una sucesión de matrices: $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$. Puesto que el cálculo de $D^{(k)}$ sólo usa a $D^{(k-1)}$, no tenemos que guardar las matrices anteriores. Al parecer, sólo necesitamos dos matrices $n \times n$. De hecho, sólo necesitamos una; el cálculo puede hacerse totalmente en la matriz D. Puesto que los elementos de la matriz sólo pueden disminuir, si supuestamente se debe usar $D^{(k-1)}[i][k]$ pero en su lugar se accede a $D^{(k)}[i][k]$, tendremos $D^{(k)}[i][k] \leq D^{(k-1)}[i][k] \leq d_{ik}^{(k-1)}$, y el cálculo podría hallar un camino aún mejor.

Algoritmo 9.4 Caminos más cortos de todos los pares (de Floyd)

Entradas: W, la matriz de pesos para un grafo cuyos vértices son $v_1, \ldots, v_n$; y n.

Salidas: D, una matriz $n \times n$ tal que `D[i][j]` es la distancia de camino más corto de v_i a v_j, siempre que el grafo no tenga ciclos con peso negativo. (Si existen ciclos con peso negativo, no estará definido un camino más corto para algunos pares de vértices; los caminos pueden acortarse dando vueltas por ese ciclo un número arbitrario de veces.) La matriz D se pasa como parámetro; el algoritmo la llena.

```
void caminosMasCortosTodosPares(float[][] W, int n, float[][] D)
    int i, j, k;
    Copiar W en D.
    for (k = 1; k <= n; k ++)
      for (i = 1; i <= n, i ++)
        for (j = 1; j <= n; j ++)
          D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
```

Es evidente que el algoritmo 9.4 efectúa $\Theta(n^3)$ operaciones.

El algoritmo puede modificarse de modo que construya una *tabla de rutas* de la cual puedan extraerse caminos más cortos, además de calcular la distancia de camino más corto. Una matriz `go` es una tabla de rutas si, para todo `go[i][j]` $= k$, existe un camino más corto de v_i a v_j cuya primera arista es (v_i, v_k). Después de llegar a k, consultamos `go[k][j]` para determinar el siguiente paso. (Véase el ejercicio 9.10.)

El problema de los caminos más cortos de todos los pares es más general que el de determinar R, la matriz de alcanzabilidad, y el algoritmo 9.4 es una generalización del algoritmo de Warshall, algoritmo 9.2. R se puede obtener de D con sólo cambiar a *verdadero* todos los elementos menores que ∞ y cambiar a *falso* todos los ∞. Para D, procesar la tripleta (i, k, j) implica calcular

```
D[i][j] = min(D[i][j], D[i][k] + D[k][j]).
```

Aquí también el orden en que se procesan las tripletas es crucial para obtener el resultado correcto sin procesamiento repetido.

9.5 Cálculo del cierre transitivo con operaciones de matrices

Supóngase que A es la matriz para una relación binaria sobre $S = \{s_1, \ldots, s_n\}$ y que interpretamos A como la relación de adyacencia sobre el grafo dirigido $G = (S, A)$. Entonces $a_{ij} = verdadero$ si y sólo si existe un camino de longitud 1 de s_i a s_j, ya que un camino de longitud 1 es una arista. Supóngase que definimos matrices $A^{(p)}$ según

$$a_{ij}^{(p)} = \begin{cases} verdadero & \text{si existe un camino de longitud } p \text{ de } s_i \text{ a } s_j \\ falso & \text{en los demás casos.} \end{cases}$$

Entonces $A^{(0)} = I$, la matriz de identidad, y $A^{(1)} = A$. ¿Cómo calculamos $A^{(2)}$? Por definición, $a_{ij}^{(2)} = verdadero$ si y sólo si existe un camino de longitud 2 de s_i a s_j, y por tanto si y sólo si existe un vértice s_k tal que $a_{ik} = verdadero$ y $a_{kj} = verdadero$. Así pues

$$a_{ij}^{(2)} = \bigvee_{k=1}^{n} (a_{ik}\, a_{kj}),$$

La fórmula para obtener $a_{ij}^{(2)}$ es la fórmula para obtener un elemento del *producto de matrices booleanas*, AA o A^2.

Definición 9.4 Operaciones de matrices booleanas

El *producto de matrices booleanas* $C = AB$ de las matrices booleanas $n \times n$ A y B es la matriz booleana cuyos elementos son

$$c_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj}) \qquad \text{para } 1 \leq i, j \leq n.$$

Las potencias de una matriz booleana se definen de la forma acostumbrada: para el entero $p \geq 0$, A^p es el producto $AA \cdots A$ (p factores).

La *suma de matrices booleanas* $D = A + B$ está definida por

$$d_{ij} = a_{ij} \vee b_{ij} \qquad \text{para } 1 \leq i, j \leq n.$$

Obsérvese que las definiciones son iguales a las del producto y la suma de matrices aritméticas, sustituyendo la suma por "$\vee$" (*or*) y la multiplicación por "$\wedge$" (*and*). ■

Con esta notación vemos que $A^{(2)} = A^2$. Es decir, A^2 indica cuáles vértices están conectados por caminos de longitud 2. Es fácil generalizar y demostrar el lema siguiente por inducción con p. La demostración se deja como ejercicio (véase el ejercicio 9.13).

Lema 9.4 Sea A la matriz booleana de adyacencia para un grafo dirigido con vértices $\{s_1, \ldots, s_n\}$. Denotemos los elementos de A^p, para $p \geq 0$, con $A^p[i][j]$. Entonces $A^p[i][j] = verdadero$ si y sólo si existe un camino de longitud p de s_i a s_j. Es decir, $A^p[i][j] = a_{ij}^{(p)}$, según la definición del principio de esta sección. □

Los elementos de R, el cierre transitivo de A, están definidos por $r_{ij} = verdadero$ si y sólo si existe un camino de *cualquier* longitud de s_i a s_j. No obstante, el lema siguiente nos permite restringir la atención a ciertos caminos; su demostración también se deja como ejercicio.

Lema 9.5 En un grafo dirigido de n vértices, si existe un camino del vértice v al vértice w, existe un camino simple de v a w, que necesariamente tiene una longitud de cuando más $n - 1$. □

Por tanto, sólo necesitamos identificar los caminos de longitud hasta $n - 1$ para obtener el cierre transitivo. Obsérvese que, para cualesquier p y q, el (i, j)-ésimo elemento de la matriz $A^p + A^q$ es *verdadero* si y sólo si existe un camino de longitud p *o* un camino de longitud q de s_i a s_j. Así pues

$$R = \sum_{p=0}^{n-1} A^p. \tag{9.4}$$

El cálculo directo de esta fórmula efectuaría $n - 2$ multiplicaciones de matrices booleanas, para obtener $A^2, A^3, \ldots, A^{n-1}$. Cada multiplicación tarda un tiempo que está en $\Theta(n^3)$ (si se usa el método directo), así que el tiempo total está en $\Theta(n^4)$. Sin embargo, hay un método mucho mejor que el directo.

Primero, observamos que no hay problema si sustituimos el límite superior de $(n - 1)$ de la ecuación (9.4) por algún valor $s \geq n - 1$. Los términos adicionales denotan caminos de longitud n o mayor, así que no son caminos simples y no conectan pares de nodos que no estén ya identificados en R. Pero, ¿en qué nos ayuda elevar el límite superior? ¿No implica simplemente más trabajo?

Una idea clave es que los exponentes que son potencias de 2 se pueden calcular elevando repetidamente al cuadrado, en lugar de subir de una en una potencia. Así, podríamos calcular A^{32} con cinco multiplicaciones, calculando A^2, luego A^4, luego A^8, y así. En cambio, A^{31} requeriría muchas más que cinco. Por tanto, podemos obtener *ciertas* potencias altas rápidamente, pero necesitamos *todas* las potencias hasta $n - 1$.

La segunda idea clave implica algunas manipulaciones algebraicas de la fórmula de la ecuación (9.4) para expresarla en una forma que sugiera un cálculo más eficiente. Nos serán útiles algunas de las propiedades siguientes de las operaciones con matrices booleanas. Supóngase que A, B y C son matrices booleanas $n \times n$.

Absorción de $+$:	$A + A = A$.
Conmutatividad de $+$:	$A + B = B + A$.
Asociatividad de $+$ y $\times$:	$A + (B + C) = (A + B) + C, A(BC) = (AB)C$.
Distributividad de $+$ sobre $\times$:	$A(B + C) = (AB) + (AC), (B + C)A = (BA) + (CA)$.
Identidad multiplicativa:	$IA = AI = A$.

Ahora, sea s la potencia más baja de 2 tal que $s \geq n - 1$. Entonces se cumple también la ecuación siguiente para R.

$$R = \sum_{p=0}^{s} A^p = I + A + A^2 + \cdots + A^s.$$

La segunda idea clave consiste en sustituir la suma de muchas potencias por una potencia de una sola matriz.

Intuitivamente, supóngase que tenemos una matriz booleana que nos da información acerca de todos los caminos de longitud 0-k. Si la multiplicamos por sí misma, nos dará información acerca de cualquier camino que pueda formarse combinando un camino de cualquier longitud 0-k con otro camino de cualquier longitud 0-k. Esto nos da todos los caminos de longitud 0-2k. Para comenzar, $(I + A)$ nos da información acerca de todos los caminos de longitud 0-1. Luego elevamos repetidamente la matriz al cuadrado hasta cubrir por lo menos los caminos de longitud $n - 1$ o menor. (Si nos fijamos, veremos que esto es casi lo que hace el algoritmo 9.1 dentro de su ciclo **`while`**.) La suma de I a A es lo que impide perder los caminos más cortos conforme aumentan las potencias. El lema y el teorema que siguen formalizan esta intuición.

Lema 9.6 $I + A + A^2 + \cdots + A^s = (I + A)^s$, donde A es una matriz booleana y $s \geq 0$.

Demostración La demostración es por inducción con s. El caso base es $s = 0$, en cuyo caso ambos miembros son iguales a I. Para $s > 0$, suponemos que $I + A + \cdots + A^{s-1} = (I + A)^{s-1}$, es decir, la igualdad del lema se cumple para $s - 1$. Entonces

$$(I + A)^s = (I + A)^{s-1} (I + A) = (I + A)^{s-1} I + (I + A)^{s-1} A.$$

Utilizando la hipótesis inductiva,

$$(I + A)^{s-1} I = I + A + \cdots + A^{s-1},$$

$$(I + A)^{s-1} A = A + A^2 \cdots + A^s.$$

Sin embargo, $A^i + A^i = A^i$ por la propiedad de absorción, de lo que se sigue la conclusión del lema. □

Teorema 9.7 Sea A una matriz booleana $n \times n$ que representa una relación binaria. Entonces R, la matriz que representa el cierre transitivo de A, es $(I + A)^s$ para cualquier $s \geq n - 1$. □

Aunque el teorema se cumple para muchos s, como ya dijimos, escogemos s de modo que sea la potencia de 2 más baja tal que $s \geq n - 1$. ¿Cuánto trabajo se requiere para calcular R utilizando la fórmula del teorema 9.7? El cálculo de $I + A$ requiere copiar A e insertar *verdadero* en la diagonal de A, lo que equivale a $\Theta(n^2)$ operaciones. Luego se puede calcular $(I + A)^s$ efectuando $\lg s = \lceil \lg(n - 1) \rceil$ multiplicaciones de matrices booleanas.

Un producto de matrices booleanas puede calcularse, como indica su definición, en tiempo $\Theta(n^3)$. No obstante, en la sección 12.3.4 veremos que es posible efectuar una multiplicación de matrices *de enteros* en tiempo $o(n^3)$ (por ejemplo, empleando el algoritmo de Strassen); el orden asintótico es aproximadamente $\Theta(n^{2.81})$. (El exponente real es lg 7; 2.81 representa su valor aproximado.) Otra alternativa para multiplicar matrices booleanas es convertirlas en matrices de enteros sustituyendo *verdadero* por 1 y *falso* por 0. Luego usamos un algoritmo para multiplicar matrices de enteros en $o(n^3)$ y por último convertimos el resultado en una matriz booleana sustituyendo todos los elementos positivos por *verdadero* y todos los ceros por *falso*. Así, R puede calcularse aproximadamente en tiempo $\Theta(n^{2.81} \log n)$. Por tanto, R puede calcularse (asintóticamente) en tiempo menor que $\Theta(n^3)$.

Ninguno de los algoritmos para obtener el cierre transitivo que hemos examinado tiene el mismo orden asintótico que los algoritmos de multiplicación de matrices más rápidos (asintóticamente). No obstante, se conoce un algoritmo de cierre transitivo, ideado por I. Munro, que es

aproximadamente 32 veces más costoso que una multiplicación de matrices booleanas del mismo tamaño, pero tiene el mismo orden asintótico. Se trata de una aplicación compleja del método de divide y vencerás. (Véase Notas y referencias al final del capítulo.)

La multiplicación de matrices booleanas es un problema más especializado que la multiplicación de matrices con elementos reales, y vale la pena buscar algoritmos especializados. En la sección que sigue desarrollaremos un algoritmo rápido para multiplicar matrices de bits.

9.6 Multiplicación de matrices de bits: algoritmo de Kronrod

Usaremos la terminología de la definición 9.3 en toda esta sección. Sean A y B matrices booleanas $n \times n$ cuyos elementos ocupan cada uno un bit. Recordemos que `A[i]` denota la i-ésima fila de A y es una cadena de bits. Utilizando la instrucción *or* por bits, se puede calcular el producto $C = AB$ como sigue, donde `C[i]` y `B[k]` son la i-ésima fila de C y la k-ésima fila de B, respectivamente.

```
Inicializar C con la matriz cero (todos sus elementos falso).
for (i = 1; i <= n; i ++)
  for (k = 1; k <= n; k ++)
    if (a_ik == true)
      ORporbits(C[i], B[k], n);
```

(Compárese esto con el algoritmo 9.3, donde se definió `ORporbits`; obsérvese la similitud de los procedimientos, a pesar de que calculan cosas distintas.) Podríamos considerar que la operación *or* por bits efectúa una unión de conjuntos. Es decir, si consideramos que `A[i]` es el conjunto $\{k \mid a_{ik} = \textit{verdadero}\}$ (un subconjunto de $\{1, 2, \ldots, n\}$), y lo mismo para las filas de B y C, entonces

$$C[i] = \cup_{k \in A[i]} B[k].$$

El algoritmo anterior efectúa cuando más n^2 uniones de filas (cada una de las cuales podría requerir varias instrucciones *or* por bits de máquina). Deduciremos un algoritmo que efectúa menos uniones de filas. El algoritmo que presentamos a continuación se conoce como el algoritmo de los Cuatro Rusos, aunque al parecer fue obra de M. A. Kronrod, uno de los cuatro.

9.6.1 Algoritmo de Kronrod

Ciertos grupos de filas de B podrían aparecer en las uniones de varias filas distintas de C. Por ejemplo, supóngase que A es la matriz que se muestra en la figura 9.4. Entonces, $B[1] \cup B[3] \cup B[4]$ está contenido en las filas 1, 3 y 7 del producto, y se efectúan nueve uniones cuando bastarían tres. ¿Cómo puede reducirse una parte de este trabajo repetido, o todo? El enfoque obvio es calcular primero muchas uniones de cantidades pequeñas de filas de B (como $B[1] \cup B[3] \cup B[4]$) y luego combinarlas de manera apropiada para obtener las filas del producto. De inmediato surgen varias preguntas:

1. ¿Cuántas filas de B, y cuáles, se deben combinar en el primer paso?
2. ¿Cómo se pueden almacenar esas uniones de modo que se pueda acceder a ellas de forma eficiente durante el segundo paso?
3. ¿Cuánto espacio adicional se necesita?
4. ¿Se ahorrará tiempo realmente en el peor caso? Si así es, ¿cuánto?

Las respuestas a la mayor parte de estas preguntas dependen de la respuesta a la primera.

$$
\begin{array}{c}
A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \\ A_8 \\ A_9 \\ A_{10} \\ A_{11} \\ A_{12}
\end{array}
\begin{bmatrix}
1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & & & & & & \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & 0 & 1 & 0 & 1 & & & & \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & &
\end{bmatrix}
$$

Figura 9.4 Una matriz de bits

Adoptaremos una estrategia directa: dividir las filas de B en varios grupos de t filas cada uno y calcular todas las posibles uniones dentro de cada grupo. Haremos caso omiso de todos los detalles de implementación hasta ver si, con valor apropiado de t, la estrategia puede dar pie a un algoritmo que efectúe menos de n^2 uniones de filas.

Sea $g = \lceil n/t \rceil$, el número de grupos que usaremos. Las filas de B se agrupan así:

$$
\begin{array}{ll}
\text{Grupo 1:} & B[1], \ldots, B[t] \\
\text{Grupo 2:} & B[t+1], \ldots, B[2t] \\
\quad \vdots & \\
\text{Grupo } g\text{:} & B[(g-1)t+1], \ldots, B[n].
\end{array}
$$

Ejemplo 9.4 Grupos de bits

Supóngase que la matriz A de la figura 9.4 se multiplicará por una matriz B de 12×12, y sea $t = 4$. Se calcularían una vez uniones de todas las combinaciones de filas $B[1]$, $B[2]$, $B[3]$ y $B[4]$. Si se efectúan en el orden correcto (primero todas las combinaciones de dos filas, luego de tres y por último de las cuatro), se pueden obtener todas las uniones efectuando 11 operaciones de unión de filas. Se haría lo mismo con los grupos $B[5]$, ..., $B[8]$ y $B[9]$, ..., $B[12]$. Entonces sólo se necesitarían dos operaciones más de unión de filas para obtener la primera fila de AB: esas operaciones calcularían

$$(B[1] \cup B[3] \cup B[4]) \cup (B[6] \cup B[8]) \cup (B[12]).$$

El valor de $B[1] \cup B[3] \cup B[4]$ se usa otra vez en la tercera y la séptima filas, y $B[6] \cup B[8]$ se usa otra vez en la sexta fila del producto. ■

Estimaremos aproximadamente el número total de uniones efectuadas en función de t y luego veremos si podemos escoger un valor de t que dé un total menor que n^2. Para cada grupo de filas (salvo quizá la última) hay que combinar 2^t conjuntos de filas. No se necesitan uniones para calcular el conjunto vacío ni los conjuntos que consisten en una sola fila (con lo que se eliminan

$t + 1$ uniones). Puesto que podemos calcular cada unión de filas dentro de un grupo combinando con una fila más conjuntos que ya se calcularon, se efectuará un total de $2^t - (t + 1)$ operaciones de unión para cada grupo. Hay g grupos, así que se efectúan $g(2^t - (t + 1))$ uniones en la primera fase del algoritmo propuesto. Ahora es posible obtener cualquier unión de filas de B deseada calculando la unión de cuando más una combinación de cada uno de los grupos. Por consiguiente, el cálculo de cada fila de la matriz producto requiere cuando más $g - 1$ uniones adicionales, o cuando más $n(g - 1)$ uniones adicionales para las n filas. El total de uniones efectuadas por este método es cuando más $g(2t - (t + 1)) + n(g - 1)$. Para simplificar nuestra tarea, aproximaremos y sólo consideraremos los términos de orden alto:

$$g(2^t - (t + 1)) + n(g - 1) \approx \frac{n\,2^t}{t} + \frac{n^2}{t}. \tag{9.5}$$

Si $t = 1$ o $t = n$, esta expresión estará en $\Theta(n^2)$ o $\Theta(2^n)$, respectivamente. Supóngase que tratamos de reducir al mínimo el miembro derecho de la ecuación (9.5) bajo el supuesto de que el primer término es de orden más alto que el segundo. Querríamos hacer a t lo más pequeño posible, pero si $t < \lg n$ el primer término ya no dominaría. Asimismo, si suponemos que el segundo término es de orden más alto, querríamos que t fuera lo más grande posible, pero no puede ser mayor que $\lg n$. Este argumento nada riguroso sugiere probar $t \approx \lg n$. El número de uniones efectuadas con $t \approx \lg n$ es aproximadamente $2n^2/\lg n$, que es de menor orden que n^2. Por tanto, vale la pena investigar este enfoque con $t \approx \lg n$. Puesto que 2^t es el término que más rápidamente crece, usamos $t = \lfloor \lg n \rfloor$. Ahora precisaremos algunos detalles de la implementación y determinaremos qué tanto espacio extra se necesita.

Por cada grupo de filas de B hay que almacenar $2^t = 2^{\lfloor \lg n \rfloor}$ conjuntos que almacenar. Primero almacenaremos los conjuntos de todos los grupos en un arreglo bidimensional de cadenas de bits, por sencillez. Más adelante veremos cómo almacenar sólo los conjuntos del grupo actual, con lo que ahorraremos algo de espacio. Los conjuntos de todos los grupos se almacenan en el arreglo `todasUniones` según el esquema siguiente, donde `todasUniones[j][i]` contiene el conjunto i para el grupo j. Las filas de B que están en el grupo j tienen índices $(j - 1)t + 1$ hasta jt. Interpretamos el segundo índice de `todasUniones` como un número binario de t bits $b_1 b_2 \cdots b_t$. Los bits de un índice i indican qué filas de B dentro del grupo j están incluidas en la unión que se almacena en `todasUniones[j][i]`; en particular, `B[(j-1)*t+k]` está incluida si y sólo si el bit b_k de i es 1.

Así, el grupo 1 de uniones se almacena como sigue:

i	Contenido de `todasUniones[1][j]`
00 . . . 00	$\varnothing$
00 . . . 01	$B[t]$
00 . . . 10	$B[t - 1]$
00 . . . 11	$B[t - 1] \cup B[t]$
⋮	⋮
11 . . . 11	$B[1] \cup B[2] \cup \cdots \cup B[t]$

Se usan exactamente 2^t celdas (en cada una de las cuales cabe una fila de B) para almacenar las uniones de cada grupo de filas. Por ahora, podemos suponer que las uniones de los demás gru-

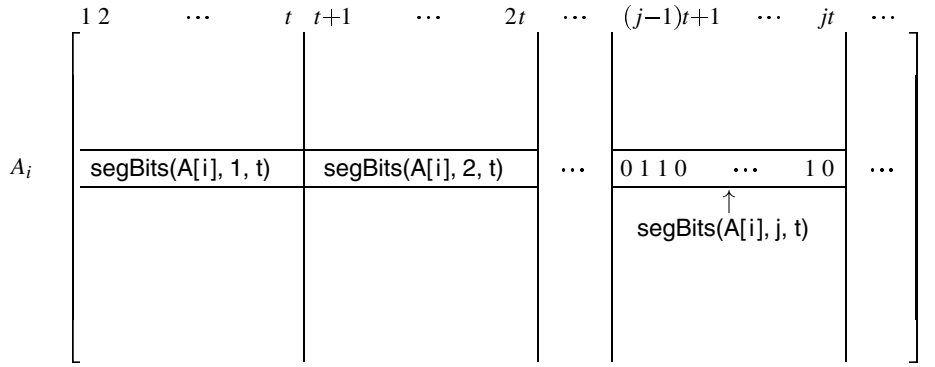


Figura 9.5 El *j*-ésimo segmento de *t* bits de la fila *i* de la matriz de bits *A*

pos se almacenan en bloques de celdas cuyo primer índice es su número de grupo. Más adelante mostraremos cómo arreglárnoslas con sólo 2^t (aproximadamente n) celdas, en vez de usar $2^t g$, o aproximadamente $n^2/\lg n$.

Ideamos este esquema de almacenamiento para que fuera más fácil hallar las uniones que se necesitan para una fila dada del producto. Recordemos que la *i*-ésima fila del producto es $\cup_{k \in A[i]} B[k]$. Supóngase que dividimos cada fila de A en segmentos de t elementos cada uno, con la notación siguiente:

Definición 9.5 Segmentos de *t* bits dentro de una cadena de bits

Sea b una cadena de bits. La subrutina `segBits(b, j, t)` devuelve el *j*-ésimo segmento de t bits, contando a partir del índice 1; es decir, se devuelven los bits $(j-1)t+1$ hasta jt como entero de t bits, donde el bit jt de b pasa a ser el bit menos significativo del entero (véase la figura 9.5). ■

Interpretado como número binario, `segBits(A[i], j, t)` es el segundo índice correcto en el arreglo `todasUniones` para la unión de filas de B pertenecientes al *j*-ésimo grupo. Por ejemplo, con la matriz de la figura 9.4, `segBits(A[7], 1, 4)` es 11 en decimal, o 1011 en binario.

Hasta aquí, el algoritmo que hemos desarrollado tiene este aspecto:

`t` $= \lfloor \lg n \rfloor$; `g` $= \lceil n/t \rceil$:
Calcular y almacenar en `todasUniones` uniones de todas las combinaciones de filas de B dentro de cada grupo de t filas sucesivas.

```
// i indiza las filas de A y C.
// j indiza grupos de filas de B.
for (i = 1; i <= n; i ++)
    Inicializar C[i] con 0.
    for (j = 1; j <= g; j ++)
      C[i] = C[i] ∪ todasUniones[j][segBits(A[i], j, t)];
```

La cantidad de espacio empleada para almacenar las uniones se puede reducir con sólo cambiar el orden en el que se trabaja. En su forma actual el algoritmo calcula una *fila* completa de C antes de pasar a la siguiente, por lo que es preciso tener a la mano todos los grupos de uniones. En cam-

bio, si el algoritmo trabaja con un *grupo* a la vez, seleccionando de ese grupo la unión que necesita para cada fila de C, grupos sucesivos de uniones podrían usar las mismas posiciones de memoria.

Los últimos dos detalles que falta precisar son un esquema eficiente para calcular las uniones dentro de cada grupo y una forma de manejar el caso en el que el último grupo tiene menos de t filas. Dejaremos el segundo problema como ejercicio. El primero se resuelve fácilmente en la forma final del algoritmo, que ahora usa un arreglo unidimensional `uniones` que tiene 2^t elementos.

Algoritmo 9.5 Multiplicación de matrices de bits (de Kronrod)

Entradas: A, B y n, donde A y B son matrices de $n \times n$ bits. (En el pseudocódigo suponemos que ya se definió la clase `MatrizDeBits`. Sus elementos comienzan en el índice 0, aunque algunas matrices de bits podrían no usar esa fila.)

Salidas: C, el producto de matrices booleanas. La matriz se pasa como parámetro y el algoritmo la llena.

Comentarios: `A[i]` y `C[i]` son las filas i-ésima de A y C. Tal como está escrito, el algoritmo supone que n es un múltiplo exacto de t. La subrutina `ORporbits` se definió en la definición 9.3, e implementa la "unión de filas". La subrutina `segBits` se definió en la definición 9.5.

```
void kronrod(MatrizDeBits A, MatrizDeBits B, int n, MatrizDeBits C)
   int t, g, i, j, k;
   t = ⌊lg n⌋; g = ⌈n/t⌉;
   MatrizDeBits uniones = new MatrizDeBits();
   Inicializar C con la matriz cero.
   for (j = 1; j <= g; j ++)

      // Calcular todas las uniones dentro del j-ésimo grupo de fi-
         las de B.
      uniones[0] = 0;
      for (k = 0; k <= t - 1; k ++)
        for (i = 0; i <= 2^k - 1; i ++)
          Copiar uniones[i] en uniones[i + 2^k].
          ORporbits(uniones[i + 2^k], B[j*t-k], n);

      // Seleccionar la unión apropiada para cada fila de C.
      for (i = 1; i <= n; i ++)
        ORporbits(C[i], uniones[segBits(A[i], j, t)], n);
      // Continuar el ciclo de j.
```

Análisis

Obsérvese que se efectúan 2^{t-1} operaciones de unión para obtener todas las uniones dentro de un grupo (en el ciclo **for** k, **for** i). El algoritmo 9.5 efectúa $(n/t)(2^t - 1 + n)$ uniones de filas en total, lo cual es menor que $2n^2/\lg(n)$ si $n > 8$. (Véase el ejercicio 9.17, donde se sugiere una posible mejora en la selección de t que reduce el coeficiente de la izquierda a 1.) El número de uniones de filas está en $\Theta(n^2/\log n)$ en cualquier caso. En la sección 9.6.2 deduciremos una cota inferior con el mismo orden asintótico para una clase de algoritmos que multiplican matrices de bits efectuando uniones de filas.

Las uniones de filas se implementan con la subrutina `ORporbits`. Esta subrutina requiere $\lceil n/w \rceil$ instrucciones *or* por bits (donde w es el tamaño de palabra, o el tamaño del operando de la instrucción *or* por bits), así que el tiempo de ejecución está en $\Theta(n^3/\log n)$, pero es un múltiplo relativamente pequeño de $n^3/\lg n$. El tiempo de ejecución no depende de la entrada; se efectúan las mismas operaciones con todas las entradas de tamaño n. El espacio extra que se requiere para el arreglo `uniones` está en $\Theta(n^2)$ bits.

La fórmula que dedujimos en la sección 9.5 para la matriz del cierre transitivo de una relación (teorema 9.7) utiliza aproximadamente $\lg n$ multiplicaciones de matrices booleanas. Por consiguiente, si usamos el algoritmo de Kronrod podremos calcular el cierre transitivo con sólo $\Theta(n^2)$ uniones de filas.

Cabe señalar que tanto el algoritmo de Warshall para el cierre transitivo (sección 9.3) como el algoritmo de Kronrod para matrices booleanas ahorran tiempo o espacio efectuando sus cálculos en un orden específico. En ambos casos el orden natural, o acostumbrado, en que se nos ocurriría realizar el trabajo es menos eficiente.

⋆ 9.6.2 Una cota inferior para la multiplicación de matrices de bits

¿Es óptimo el algoritmo de Kronrod? Si consideramos el tiempo que toma efectuar las uniones de filas, no lo es; tarda un tiempo $\Theta(n^3/\log n)$, y el orden del algoritmo de Strassen, $n^{2.81}$, es un orden asintótico menor. Los diversos algoritmos para multiplicar matrices booleanas suponen diferentes representaciones para las matrices (matrices de bits *vs*. un elemento por palabra) y efectúan distintos tipos de operaciones (por ejemplo, operaciones booleanas con palabras, operaciones aritméticas si se usa el método de Strassen, o uniones de filas como en el algoritmo de Kronrod). Si restringimos nuestra atención a la clase de algoritmos que calculan filas del producto de las matrices de bits formando uniones de filas de la segunda matriz factor, podremos demostrar que, dentro de esta clase, el algoritmo de Kronrod tiene orden asintótico óptimo: el número de uniones efectuadas por un algoritmo óptimo también estaría en $\Theta(n^2/\log n)$.

Uno de los motivos para incluir la demostración del teorema es que ilustra un "argumento de conteo", un enfoque útil para establecer cotas inferiores que implica contar todos los posibles algoritmos (sin tomar en cuenta diferencias que no sean pertinentes a la sucesión de operaciones básicas: en este caso, uniones de filas efectuadas por los algoritmos).

Para deducir la cota inferior usamos un modelo de algoritmos abstraído (como hicimos con los árboles de decisión para ordenamiento). Sea **A** un algoritmo que calcula $C = AB$ formando uniones de filas de B (y posiblemente copiando filas) y no puede ejecutar otras operaciones con B. Para una entrada específica, A y B, podemos preparar una lista indizada de las operaciones de unión efectuadas por **A**, denotando tal operación con `union`(r, s), donde r y s podrían ser una fila de B o el resultado de una `union` anterior especificada por su índice en la lista.

La sucesión de operaciones `union` efectuadas no basta para describir el resultado que el algoritmo produce; es preciso saber cuáles de las uniones calculadas en la sucesión van a ser filas del producto y qué filas del producto son. Supóngase que A y B son $n \times n$, y sea `pasos` el número de pasos en la lista de operaciones `union`. Entonces la información adicional que se requiere la puede proporcionar un vector-n $V = (j_1, \ldots, j_n)$, donde $-n \leq j_i \leq$ `pasos` y j_i describe la i-ésima fila de la matriz producto C, como sigue: si $j_i > 0$, la i-ésima fila es el resultado de la j-ésima operación `union`; si $j_i = 0$, la i-ésima fila sólo contiene ceros (el conjunto vacío); y si $j_i < 0$, la i-ésima fila es la $|j_i|$-ésima fila de B.

Ejemplo 9.5 Uniones de filas y el vector V

Si

$$A = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

un algoritmo podría efectuar la sucesión de operaciones de unión y copiado siguientes:

```
1. tmp1 = union(B[1], B[4]);
2. C[1] = union(tmp1, B[2]);
3. C[2] = union(tmp1, B[3]);
4. C[3] = union(C[1], C[2]);
5. C[4] = B[4];
```

El vector V para este ejemplo es $(2, 3, 4, -4)$. ■

Teorema 9.8 Si n es lo bastante grande (en particular, si $n > 1024$), cualquier algoritmo que multiplique matrices booleanas utilizando uniones de filas deberá efectuar por lo menos $n^2/5 \lg(n)$ operaciones de unión para multiplicar matrices $n \times n$ en el peor caso.

Demostración Sea **A** un algoritmo que calcula $C = AB$, y supóngase que **A** efectúa cuando más $2n^2/\lg n$ uniones de filas con $n > 1024$. Sea $F(n)$ el número de uniones efectuadas por **A** para multiplicar una matriz $n \times n$ arbitraria, A, y la matriz de identidad, I_n, en el peor caso. El número de uniones efectuadas por **A** en el peor caso con cualquier entrada es de por lo menos $F(n)$, y cualquier cota inferior que se obtenga para $F(n)$ será una cota inferior para cualquier algoritmo de la clase considerada. Demostraremos que $F(n) \geq n^2/5 \lg n$, si n es lo bastante grande. (Omitiremos los pormenores de la demostración de que 1024 es lo bastante grande, pero es fácil incluirlos mediante cálculos directos.)

Sea S_n el conjunto de todas las sucesiones válidas de $F(n)$ operaciones de unión. (Una sucesión es válida si, para cada i, la i-ésima operación se refiere a filas de B que están entre la 1 y la n y/o a los resultados de operaciones con índices entre 1 e $i - 1$.) Sea V_n el conjunto de todos los vectores-n con elementos enteros entre $-n$ y $F(n)$. Las operaciones efectuadas por **A** y la salida de A con una entrada A dada están descritas por un elemento de $S_n \times V_n$. Si **A** efectúa menos de $F(n)$ uniones con una A dada, S_n contendrá una sucesión que efectúa el trabajo de **A** y luego se rellena hasta la longitud $F(n)$ con repeticiones de, digamos, `union(1, 1)`. Deduciremos una cota superior y una cota inferior para $|S_n \times V_n|$ y usaremos la desigualdad resultante para obtener una cota inferior de $F(n)$.

Puesto que cada `union` tiene dos operandos, cada uno de los cuales es una fila de B o un índice entre 1 y $F(n)$, hay $(n + F(n))^2$ opciones para cada operación `union`. Por tanto, $|S_n| \leq (n + F(n))^{2F(n)}$. $|V_n| = (n + 1 + F(n))^n$, así que $|S_n \times V_n| \leq (n + 1 + F(n))^{2F(n)+n}$.

Para obtener una cota inferior de $|S_n \times V_n|$, observamos que $S_n \times V_n$ contiene un elemento distinto por cada matriz A de $n \times n$, puesto que $A_1 I_n \neq A_2 I_n$ si $A_1 \neq A_2$. Por tanto, $|S_n \times V_n| \geq 2^{n2}$, ya que hay 2^{n2} matrices booleanas $n \times n$. Por tanto,

$$2^{n^2} \leq \left| S_n \times V_n \right| \leq (n + 1 + F(n))^{2F(n)+n}$$

o bien

$$n^2 \leq (2\ F(n) + n)\ \lg(n + 1 + F(n)) \qquad \text{para todo } n > 0. \tag{9.6}$$

Observamos que $F(n) > n^{3/2}$ si n es lo bastante grande, porque si no lo es la ecuación (9.6) implicaría que n^2 está en $O(n^{3/2} \log n)$, y esto no es cierto. Puesto que $F(n) > n^{3/2}$, $2F(n) + n < 2.1\ F(n)$ si n es lo bastante grande.

Además, $F(n) \leq 2n^2/\lg n$ (por la selección de **A**), así que $n + 1 + F(n) < 2n^2$ si n es lo bastante grande. La sustitución de estas desigualdades en la ecuación (9.6) da

$$n^2 \leq 2.1\ F(n)\ \lg(2n^2) = 2.1\ F(n)(1 + 2\ \lg n) \qquad \text{si } n \text{ es lo bastante grande.}$$

También tenemos $1 + 2 \lg n \leq 2.1 \lg n$ si n es lo bastante grande, así que

$$n^2 \leq 2.1^2\ F(n)\ \lg n \qquad \text{si } n \text{ es lo bastante grande.}$$

Pero $2.1^2 < 5$, así que $n^2 < 5\ F(n) \lg n$, o $F(n) > n^2/5 \lg(n)$, si n es lo bastante grande. □

Ejercicios

Sección 9.2 Cierre transitivo de una relación binaria

9.1 Sea $G = (V, E)$ un grafo no dirigido y sea R una relación sobre V definida por vRw si y sólo si existe un camino de v a w. (Recuerde que hay un camino de longitud cero de cualquier vértice a sí mismo.)

a. Demuestre que R es una relación de equivalencia.

b. Determine las clases de equivalencia de esta relación.

c. Muestre que la matriz de alcanzabilidad R para un grafo no dirigido con n vértices se puede construir en tiempo $O(n^2)$.

9.2

a. Trate de escribir un algoritmo empleando búsqueda primero en profundidad para construir R, la matriz de alcanzabilidad de un grafo dirigido, dada A, la matriz de adyacencia. El algoritmo deberá usar la sugerencia de la sección 9.2.2, de que se calculen los elementos de R de varias filas durante una búsqueda primero en profundidad. Utilice cualquier otro ardid que se le ocurra para diseñar un algoritmo eficiente.

b. ¿Qué orden asintótico tiene el tiempo de ejecución de peor caso de su algoritmo?

c. Pruebe su algoritmo con el digrafo de la figura 9.6. Si no funciona correctamente, modifíquelo para que lo haga y repita la parte (b).

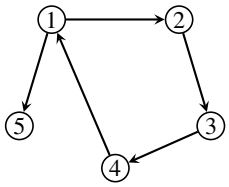


Figura 9.6 Grafo dirigido para el ejercicio 9.2

Sección 9.3 Algoritmo de Warshall para cierre transitivo

9.3 Utilice el algoritmo 9.2 para calcular el cierre transitivo de la relación A dada en el ejemplo 9.1. Muestre la matriz después de cada pasada por el ciclo **`for`** más exterior.

9.4 Construya el peor ejemplo que pueda para el algoritmo 9.1, es decir, un ejemplo con el que el ciclo **`for`** triple se repita muchas veces. ¿Cuántas veces se repetirá el ciclo en su ejemplo?

9.5 Utilice el algoritmo 9.3 para calcular el cierre transitivo de la relación A dada en el ejemplo 9.1. Especifique qué operaciones `ORporbits` se efectúan y demuestre sus resultados. Muestre también la matriz después de cada pasada por el ciclo **`for`** más exterior.

Sección 9.4 Caminos más cortos de todos los pares en grafos

9.6 Construya un ejemplo de grafo dirigido ponderado con el que el algoritmo 9.4 no funcionaría correctamente si k se variara en el ciclo más interior en vez de en el más exterior.

9.7 Utilice el algoritmo 9.4 para calcular la matriz de distancias del grafo dirigido cuya matriz de adyacencia es

$$\begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & 3 \\ \infty & 1 & 4 & 0 \end{bmatrix}$$

9.8

a. Utilice el algoritmo 9.4 para calcular la matriz de distancias del grafo dirigido cuya matriz de adyacencia es

$$\begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & -3 \\ \infty & -1 & 4 & 0 \end{bmatrix}$$

b. Explique por qué este algoritmo funciona correctamente aunque algunos de los pesos sean negativos, en tanto no haya ciclos negativos. (Un ciclo negativo es un ciclo en el que la suma de los pesos de las aristas es negativa.)

9.9 Demuestre el lema 9.3.

9.10 Indique cómo modificar el algoritmo 9.4 de modo que construya una *tabla de rutas*, la cual se describe en el texto después del algoritmo. Llame `go` a la matriz para la tabla de rutas. *Sugerencia:* Si se está actualizando `D[i][j]` porque se halló un camino más corto, y ese camino pasa por el vértice intermedio k, ¿cuál sería el primer paso de ese camino?

9.11 Calcule la tabla de rutas `go` para el grafo ponderado del ejercicio 9.7. Cabe señalar que la forma más fácil de hacerlo es simultáneamente con el cálculo de la matriz de distancias.

9.12 Sugiera un algoritmo para determinar la longitud de un ciclo más corto en un grafo dirigido. ¿Su algoritmo también funciona con grafos no dirigidos? Explique por qué sí o por qué no.

Sección 9.5 Cálculo del cierre transitivo con operaciones de matrices

9.13 Demuestre el lema 9.4.

9.14 Demuestre el lema 9.5.

9.15 Demuestre que A^+, el cierre transitivo no reflexivo de la matriz booleana A, se puede calcular con una multiplicación de matrices si se conoce el cierre transitivo (reflexivo) A^*.

Sección 9.6 Multiplicación de matrices de bits: algoritmo de Kronrod

9.16 Demuestre que si A y B son matrices booleanas $n \times n$ cuyas filas se interpretan como subconjuntos de $\{1, 2, \ldots, n\}$, tal como se describió al principio de la sección 9.6, y si $C = AB$, entonces la i-ésima fila de C es $\cup_{k \in A[i]} B[k]$.

9.17 Analice una variación del algoritmo 9.5 en la que el tamaño de grupo es $t = \lfloor \lg(n/\lg(n)) \rfloor = \lfloor \lg(n) - \lg \lg(n) \rfloor$. ¿Cuántas uniones se efectúan con esta variación en comparación con el valor de t empleado en el algoritmo?

Problemas adicionales

9.18 Un *triángulo* en un grafo es un ciclo de longitud 3. Bosqueje un algoritmo que utilice la matriz de adyacencia de un grafo para determinar si tiene un triángulo. ¿Cuántas operaciones con elementos de matriz efectúa su algoritmo?

Programas

1. Escriba un programa para multiplicar dos matrices de bits empleando el algoritmo de Kronrod (algoritmo 9.5). Contemple la posibilidad de que n sea mayor que el número de bits por palabra. ¿Cuánto espacio se usa?

Notas y referencias

Los algoritmos 9.2 y 9.3 aparecieron en Warshall (1962). El lector puede hallar demostraciones de la corrección del algoritmo 9.2 (teorema 9.2) y del algoritmo 9.3 ahí y en Wegner (1974). El algoritmo 9.4, para hallar distancias en grafos, apareció en Floyd (1962). El cierre semianular es una generalización de ambos problemas, se analiza en Aho, Hopcroft y Ullman (1974) y en Cormen, Leiserson y Rivest (1990). El primer algoritmo de este género podría ser el que aparece en Kleene (1956), el cual se aplica a autómatas finitos. La demostración de que el cálculo del cierre transitivo reflexivo se puede efectuar en tiempo del mismo orden que la multiplicación de matrices booleanas se debe a Munro (1971) pero también aparece en Aho, Hopcroft y Ullman (1974).

El algoritmo de Kronrod (algoritmo 9.5) apareció en Arlazarov, Dinic, Kronrod y Faradzev (1970) (donde no se habla de ninguna implementación). La demostración del teorema 9.8, la cota inferior para la multiplicación de matrices booleanas mediante uniones de filas, se basa en Angluin (1976). Este resultado, junto con generalizaciones del algoritmo de Kronrod, aparece en Savage (1974).

10

Programación dinámica

10.1 Introducción

Quienes no pueden recordar el pasado están condenados a repetirlo.

—George Santayana, *La vida de la razón; o, Fases del progreso humano* (1905)

La programación dinámica ha evolucionado hasta convertirse en un importante paradigma del diseño de algoritmos en ciencias de la computación. No obstante, para muchas personas su nombre es un tanto misterioso; fue acuñado en 1957 por Richard Bellman para describir un tipo de problema de control óptimo. En realidad, el nombre originalmente describía el problema más que la técnica para resolverlo. El sentido que se da a *programación* es el de "una serie de opciones", como la programación de una estación de radio. La palabra *dinámica* da la idea de que las opciones podrían depender del estado actual, en lugar de estar decididas con antelación. Así pues, en este sentido original, podríamos decir que un programa de radio en el que los escuchas llaman para hacer solicitudes está "programado dinámicamente" para contrastarlo con el formato más común en el que se decide qué canciones se tocarán antes de que comience el programa. Bellman describió un método de solución para problemas de "programación dinámica", que ha inspirado varios algoritmos para computadora. La característica principal de su método era que sustituía un cálculo en tiempo exponencial por un cálculo en tiempo polinómico. Ésta sigue siendo una característica de todos los algoritmos de programación dinámica.

Este capítulo difiere de casi todos los otros en cuanto a que normalmente nos concentramos en un problema o área de aplicación y consideramos diversos algoritmos que podríamos usar; en este capítulo, en cambio, nos concentraremos en una técnica y desarrollaremos soluciones de programación dinámica para problemas de distintas áreas de aplicación.

El diseño descendente de algoritmo es un enfoque natural y muy útil. Primero pensamos y planeamos en términos generales, y luego añadimos más y más detalles. Resolvemos un problema complejo, de alto nivel, descomponiéndolo en subproblemas. Si empleamos recursión, resolvemos un problema grande descomponiéndolo en casos más pequeños del mismo problema. Divide y Vencerás, una técnica recursiva para diseñar algoritmos, resultó especialmente útil para obtener algoritmos de ordenamiento rápidos. No obstante, pese a las bondades de la recursión, si no se le controla debidamente puede perder mucha eficiencia. Los números de Fibonacci ofrecen un ejemplo sencillo e impresionante.

Ejemplo 10.1 Función de Fibonacci recursiva

Recordemos (ecuación 1.13) que los números de Fibonacci están definidos por la recurrencia $F_n = F_{n-1} + F_{n-2}$ para $n \geq 2$, con valores de frontera $F_0 = 0$ y $F_1 = 1$. Su definición es recursiva, y es natural calcularlos con una función recursiva, `fib(n)`, como la que se dio en el ejemplo 3.1. Sin embargo, como ilustra la figura 7.13, el cálculo recursivo natural es terriblemente ineficiente porque se repite gran parte del trabajo. Básicamente, esa figura muestra el árbol de activación para `fib(6)`. En general, el árbol de activación para `fib(n)` es un árbol binario completo hasta la profundidad $n/2$ (siendo el camino de la extrema derecha el más corto), y tiene más nodos a mayores profundidades, así que el tiempo de ejecución está *por lo menos* en $\Omega(2^{n/2})$. El orden asintótico exacto es el tema del ejercicio 10.1. Sin embargo, F_n puede calcularse con $\Theta(n)$ enunciados simples calculando *y recordando n* valores más pequeños, cada uno de los cuales se puede calcular con un número constante de operaciones si se cuenta con los valores más pequeños. (Recordemos lo dicho en la sección 3.2.1, que un *enunciado simple* no incluye invocaciones

de funciones, y se supone que requiere un tiempo constante.) Si suponemos que se ha reservado espacio para un arreglo `f` del tamaño suficiente, el procedimiento que sigue cumple con el cometido:

```
f[0]=0; f[1]=1;
for (i=2; i<=n; i++)
     f[i]=f[i-1]+f[i-2];
```

En el ejercicio 10.2 el arreglo no es necesario. ■

Un algoritmo de programación dinámica almacena los resultados, o soluciones de subproblemas más pequeños y posteriormente los consulta en lugar de volver a calcularlos, cuando los necesita para resolver subproblemas más grandes. Así pues, la programación dinámica es idónea para problemas en los que un algoritmo recursivo resolvería muchos de los subproblemas varias veces.

Presentaremos una caracterización de los algoritmos de programación dinámica que ofrece un marco unificado para una amplia variedad de algoritmos publicados que a primera vista podrían parecer muy diferentes. Este marco permite convertir una solución recursiva en un algoritmo de programación dinámica, analizando la complejidad de dicho algoritmo.

10.2 Grafos de subproblema y su recorrido

Como dijimos antes, es común resolver problemas descomponiéndolos en problemas más pequeños del mismo tipo, resolviendo los problemas pequeños recursivamente y combinando después las soluciones. Supóngase que estamos contemplando un método de resolución de este tipo. Podemos definir un grafo dirigido con base en las relaciones entre los problemas y sus subproblemas pertinentes.

Definición 10.1 Grafo de subproblemas

Supóngase que se conoce un algoritmo recursivo **A** para resolver un problema. El *grafo de subproblemas para* **A** es el grafo dirigido cuyos vértices son los casos, o entradas, de este problema y cuyas aristas dirigidas son $I \rightarrow J$ para todos los pares tales que, cuando el algoritmo **A** se invoca con el caso I del problema, efectúa una invocación recursiva (directamente) con el caso J. (Aquí usamos la notación "$I \rightarrow J$" en lugar de "IJ" para hacer hincapié en que las aristas tienen dirección.) A diferencia de la mayor parte de los grafos que hemos considerado hasta ahora, que se representaron explícitamente con una estructura de datos, este grafo es abstracto y no tiene una representación explícita.

Sea P un caso del problema para el algoritmo **A**; es decir, suponemos que **A**(P) no es una invocación recursiva. Entonces el *grafo de subproblemas para* **A**(P) es la porción del grafo de subproblemas para **A** a la que se puede llegar desde el vértice P. ■

Ejemplo 10.2 Grafo de subproblemas para la función de Fibonacci

Para la función de Fibonacci recursiva, `fib(n)`, los casos del problema son simplemente los enteros no negativos, así que éstos son los vértices del grafo de subproblemas para F. Las aristas dirigidas son $\{i \rightarrow i-1 \mid i \geq 2\} \cup \{i \rightarrow i-2 \mid i \geq 2\}$. Aunque el grafo es infinito, para cualquier n dada la porción que es pertinente para el cálculo de `fib(n)` (es decir, a la que se puede

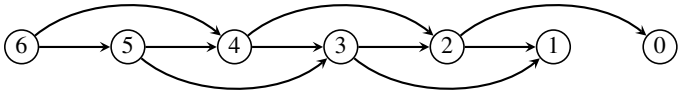


Figura 10.1 Grafo de subproblemas para `fib(6)`

llegar desde el vértice n) sólo tiene $n + 1$ vértices y aproximadamente $2n$ aristas. La figura 10.1 muestra un ejemplo. ■

Si el algoritmo **A** siempre termina, su grafo de subproblemas debe ser acíclico. Estudiamos los grafos acíclicos dirigidos (DAG) en la sección 7.4.6, y pronto tendremos oportunidad de usar algunos de esos resultados. Si examinamos el árbol de marcos de activación (sección 3.2.1) generado por una invocación de nivel más alto específica, digamos **A**(P), es evidente que cada camino del árbol corresponde a un camino del grafo de subproblemas para **A**(P) que parte del vértice P y termina en un vértice de caso base, del cual no salen aristas. Debemos tener presente que los vértices son casos del problema en este grafo abstracto. Las aristas dirigidas corresponden a invocaciones recursivas que se harían durante la ejecución de **A**(P).

Consideremos un procedimiento para recorrer grafos parecido al esqueleto de búsqueda primero en profundidad (algoritmo 7.3) pero que no colorea los vértices para recordar cuáles ya se han descubierto o terminado. Decimos que se hace un recorrido *sin memoria* del grafo. Un recorrido sin memoria de un grafo recorre todos los caminos de un grafo acíclico (y en un grafo con ciclos podría no terminar). Un cálculo recursivo natural, que simplemente efectúa invocaciones recursivas cuando necesita hacerlo, es como un recorrido sin memoria del grafo de subproblemas para **A**(P). En tanto el grafo de subproblemas sea acíclico, el procedimiento terminará tarde o temprano. No obstante, un grafo acíclico puede tener un número exponencial de caminos.

Para resumir nuestra situación, si tenemos una estrategia recursiva para resolver un problema, y P es el caso del problema que deseamos resolver, tendremos que resolverlo para todos los vértices del grafo de subproblemas a los que se puede llegar desde P. Si hay varios caminos para llegar a un subproblema, el procedimiento recursivo natural lo resolverá varias veces.

Si I es cualquier subproblema que es preciso resolver, e I tiene aristas a $J_1, J_2, \ldots, J_k$, será necesario resolver esos subproblemas antes de resolver I. En otras palabras, el grafo de subproblemas también puede verse como un *grafo de dependencia*, como en el ejemplo 7.14. Si encontramos un orden en el que programar la resolución de los subproblemas y recordamos las soluciones para usarlas después, sólo será necesario resolver una vez cada subproblema.

Como vimos en la sección 7.4.6, cualquier orden topológico inverso produce un programa aceptable para un grafo de dependencia. La esencia de la programación dinámica consiste en hallar un orden topológico inverso para el grafo de subproblemas, y asentar las soluciones de los subproblemas para que las usen posteriormente otros subproblemas.

En muchos casos, es posible determinar un orden topológico inverso con base en un conocimiento del problema. En el caso de los números de Fibonacci, dicho orden es simplemente el orden ascendente. Para algunos de los problemas que estudiaremos en secciones posteriores, el orden en cuestión es menos obvio, pero de todos modos puede deducirse a partir del conocimiento del problema. Sin embargo, en la sección 7.4.6 desarrollamos una herramienta general para hallar una numeración topológica inversa de *cualquier* DAG, el algoritmo 7.5. Este algoritmo simple-

mente ejecuta el esqueleto DFS y asigna el número topológico inverso en orden posterior. Por tanto, si no vemos alguna forma sencilla de definir un orden topológico inverso para cierto problema, podríamos dejar la tarea a este algoritmo. A continuación veremos que no es necesario hacer esto como paso independiente. Podría ser recomendable que el lector repase el esqueleto de búsqueda primero en profundidad del algoritmo 7.3 antes de continuar.

El esqueleto DFS no es en sí más que un procedimiento recursivo. Cuando lo aplicamos al grafo de subproblemas, simplemente imita el algoritmo recursivo, digamos **A**, para el problema que deseamos resolver; recordemos que el grafo de subproblemas se basa en el patrón de invocaciones recursivas que **A** efectúa. Es decir, la exploración de cada arista del esqueleto DFS corresponde a una invocación recursiva en **A**, y cuando se inicia el recorrido en orden posterior ya se ha acumulado toda la información que **A** necesita para calcular la solución. No obstante, **A** explora *todas* las aristas en el sentido de hacer la invocación recursiva, mientras que el esqueleto DFS sólo explora las aristas que conducen a vértices no descubiertos, y *verifica* las demás aristas. En síntesis, el esqueleto DFS recuerda dónde ha estado coloreando los vértices que visitó. Esta observación nos lleva a la caracterización de los algoritmos de programación dinámica.

Definición 10.2 Versión de programación dinámica de un algoritmo recursivo

Una *versión de programación dinámica* de un algoritmo recursivo **A** dado, que denotamos con $\mathcal{DP}(\mathbf{A})$, es un procedimiento que, dado un problema de nivel más alto a resolver, digamos P, efectúa una búsqueda primero en profundidad en el grafo de subproblemas para $\mathbf{A}(P)$. Conforme se obtienen soluciones para los subproblemas, se asientan en un diccionario, llamémoslo `soln`. Es decir, `soln` es un objeto de un tipo de datos abstracto `Dicc`. El proceso de asentar soluciones de subproblemas se conoce como *memo-ización*. Recordemos que las operaciones del TDA `Dicc` son `crear`, `miembro`, `recuperar` y `almacenar` (sección 2.5.3).

En general, el procedimiento de **A** se convierte en $\mathcal{DP}(\mathbf{A})$ insertando unos cuantos enunciados según el esquema siguiente. Supóngase que P es el problema actual.

1. Antes de cualquier invocación recursiva, digamos con el subproblema Q, examinar el diccionario `soln` para ver si se ha almacenado una solución para Q.
   a. Si no se ha guardado ninguna solución, efectuar la invocación recursiva, tratando así a Q como vértice blanco y tratando a $P \rightarrow Q$ como arista de árbol.
   b. Si ya se guardó una solución para Q, recuperar la solución almacenada y *no* hacer la invocación recursiva, tratando así a Q como vértice negro.
2. Justo antes de devolver la solución de P, almacenarla en el diccionario `soln`; esto tiene el efecto de colorear de negro el vértice P.

En este esquema, es indispensable que el grafo de subproblemas sea acíclico, porque los vértices no se colorean de gris, lo cual normalmente se hace para evitar el recorrido de ciclos.

Al igual que la búsqueda primero en profundidad, $\mathcal{DP}(\mathbf{A})$ requiere una "envoltura" para preparar la ejecución del procedimiento recursivo. Como mínimo, dicha envoltura crea `soln` como diccionario vacío, lo que tiene el efecto de pintar de blanco todos los vértices alcanzables del grafo de subproblemas. Este diccionario depende del problema de nivel más alto, digamos P, porque debe poder almacenar una solución para cada subproblema al que se puede llegar desde P en el grafo de subproblemas.

En muchos casos el algoritmo recursivo original, **A**, requería una envoltura, casi siempre para inicializar ciertas estructuras globales que dependen del problema de nivel más alto. En estos casos la envoltura de programación dinámica deberá incluir el procesamiento efectuado por la envoltura original, además de crear el diccionario vacío. ■

Veremos que el número de subproblemas alcanzables (y por ende el tamaño del diccionario) es un factor crítico para el diseño y análisis de algoritmos de programación dinámica eficientes.

Ejemplo 10.3 *DP*(`fib`)

La versión de programación dinámica de la función de Fibonacci `fib` sería parecida a ésta.

```
envoltFibPD(n)
    Dicc soln = crear(n);

    return fibPD(soln, n);

fibPD(soln, k)
    int fib, f1, f2;
    if (k < 2)
      fib = k;
    else
      if (miembro(soln, k−1) == false)
        f1 = fibPD(soln, k−1):
      else
        f1 = recuperar(soln, k−1);

      if (miembro(soln, k−2) == false)
        f2 = fibPD(soln, k−2);
      else
        f2 = recuperar(soln, k−2);

      fib = f1 + f2;
    almacenar(soln, k, fib);
    return fib;
```

Desde luego, en este ejemplo sencillo es fácil hallar muchas simplificaciones, que dan como resultado un algoritmo como el del ejercicio 10.1. Su propósito es ilustrar la naturaleza general de la transformación de **A** a *DP*(**A**). Cabe señalar que, así como la búsqueda primero en profundidad requiere una envoltura en torno a su procedimiento recursivo, *DP*(**A**) también requiere una envoltura. Por tanto, `envoltFibPD` inicializa un diccionario apropiado para el problema de nivel más alto (*n* en este caso) y luego invoca `fibPD(soln, n)`. ■

Incluso cuando es posible hallar un orden topológico inverso mediante inspección, el punto de vista DFS puede ser valioso para analizar la complejidad. Sabemos que la DFS procesa cada vértice una vez y cada *arista* una vez, y que normalmente hay más aristas que vértices. Si podemos asignar todo el trabajo del algoritmo a diversos vértices y aristas, ello nos podría ayudar a obtener un buen estimado del tiempo de ejecución.

10.3 Multiplicación de una sucesión de matrices

En esta sección presentaremos el problema del orden de multiplicación de matrices, que es uno de los ejemplos clásicos de programación dinámica. En la sección que sigue estudiaremos un problema que surge de una aplicación totalmente distinta pero que tiene una solución muy similar. Juntos, deberán servir como introducción excelente a la programación dinámica.

El propósito de esta sección no es mostrar la forma de resolver el problema del orden de multiplicación de matrices, sino más bien cómo aplicar los principios de desarrollo de un algoritmo de programación dinámica, paso por paso. Confiamos en que estos principios ayudarán a los lectores a resolver problemas nuevos y desarrollar una intuición respecto a los casos en que la programación dinámica es una estrategia factible. Sin embargo, el tratamiento del problema del orden de multiplicación de matrices es más complejo de lo que sería necesario si el único objetivo fuera presentar y explicar la solución de este único problema.

10.3.1 El problema del orden de multiplicación de matrices

Supóngase que nos interesa determinar el orden óptimo para efectuar multiplicaciones de matrices cuando es preciso multiplicar entre sí una serie de más de dos matrices. Utilizamos el algoritmo ordinario de multiplicación de matrices (algoritmo 1.2) cada vez que multiplicamos dos matrices. Así pues, para multiplicar una matriz de $p \times q$ y una de $q \times r$ elementos, efectuamos pqr multiplicaciones de elementos. Debemos hacer dos observaciones importantes. Primera, obtendremos el mismo resultado sea cual sea el orden en que efectuemos las multiplicaciones. Es decir, la multiplicación de matrices es *asociativa:* $A(BC) = (AB)C$. Segunda, el orden puede influir drásticamente en la cantidad de trabajo efectuada. Consideremos el ejemplo siguiente.

Ejemplo 10.4 Diversos órdenes de multiplicación

Queremos multiplicar arreglos de los tamaños que se muestran:

$$\underset{30 \times 1}{A_1} \times \underset{1 \times 40}{A_2} \times \underset{40 \times 10}{A_3} \times \underset{10 \times 25}{A_4}$$

Los cálculos siguientes muestran cuántas multiplicaciones se efectúan siguiendo diferentes órdenes.

$$\begin{array}{lrcrcrcr}
((A_1A_2)A_3)A & 30 \cdot 1 & \cdot 40 & + & 30 \cdot 40 \cdot 10 & + & 30 \cdot 10 \cdot 25 & = & 20{,}700 \\
A_1(A_2(A_3A_4)) & 40 \cdot 10 & \cdot 25 & + & 1 & \cdot 40 \cdot 25 & + & 30 \cdot 1 \cdot 25 & = & 11{,}750 \\
(A_1A_2)(A_3A_4) & 30 \cdot 1 & \cdot 40 & + & 40 \cdot 10 \cdot 25 & + & 30 \cdot 40 \cdot 25 & = & 41{,}200 \\
A_1((A_2A_3)A_4) & 1 & \cdot 40 \cdot 10 & + & 1 & \cdot 10 \cdot 25 & + & 30 \cdot 1 \cdot 25 & = & 1{,}400
\end{array}$$ ■

Para el problema general, supóngase que se nos dan las matrices $A_1, A_2, \ldots, A_n$, donde las dimensiones de A_i son $d_{i-1} \times d_i$ (para $1 \leq i \leq n$). ¿Cómo deberemos calcular

$$\underset{d_0 \times d_1}{A_1} \times \underset{d_1 \times d_2}{A_2} \times \cdots \times \underset{d_{n-1} \times d_n}{A_n}$$

y cuál es el costo mínimo de hacerlo? Nuestro costo es el número de multiplicaciones de elementos. (También podría usarse alguna otra función de costo.) Por ahora nos concentraremos en el problema de determinar el costo mínimo; más adelante haremos que el algoritmo "recuerde" cómo se obtuvo el mínimo. Denotaremos el operador de multiplicación entre A_k y A_{k+1} como la k-ésima multiplicación.

10.3.2 Un intento codicioso

Cualquier sucesión de las $n-1$ multiplicaciones es válida, y el algoritmo necesita determinar cuál sucesión tiene el costo total más bajo. El enfoque codicioso es verosímil. Primero se escoge la multiplicación de costo mínimo. Después de esta multiplicación, se determinan las dimensiones de las matrices en la cadena de matrices modificada. Se escoge una vez más la multiplicación de costo mínimo, y así. La estrategia funciona con el ejemplo 10.4, pero no logra ser óptima con algunas sucesiones de tres matrices (sólo dos multiplicaciones de matrices). En el ejercicio 10.6 exploraremos otra estrategia codiciosa. Por lo regular, los algoritmos de programación dinámica son más costosos que los codiciosos, por lo que se usan sólo cuando no se puede hallar una estrategia codiciosa que proporcione la solución óptima.

10.3.3 Hacia una solución de programación dinámica

Ahora intentaremos desarrollar un algoritmo recursivo. Supóngase que, después de escoger una primera multiplicación (digamos, en la posición i de la sucesión), resolvemos recursivamente el problema restante de manera óptima. Haremos esto para cada i que represente una primera opción válida, al final escogeremos la i que dé el costo combinado más bajo. Decimos que este algoritmo es *de retroceso* porque después de probar una sucesión de opciones completa, el algoritmo retrocede al punto en que estaba antes de la decisión más reciente y prueba una alternativa; una vez que se agotan las alternativas en ese punto, el algoritmo retrocede a un punto anterior y prueba alternativas ahí, continuando hasta agotar todas las alternativas. Vimos un ejemplo de esta idea en el problema de las ocho reinas (véase la figura 7.14).

Supóngase que las dimensiones $d_0, \ldots, d_n$ están en un arreglo `dim`. Podemos dejar intacto el arreglo y simplemente identificar un subproblema con una sucesión de enteros que dan los índices de las dimensiones de las matrices restantes. La sucesión inicial de índices es $0, \ldots, n$. Cabe señalar que todos los índices de la sucesión, con excepción del primero y el último, especifican también operadores de multiplicación.

Después de escoger como primera opción la multiplicación i, la sucesión de índices para el problema restante es $0, \ldots, i-1, i+1, \ldots, n$. Es decir, la primera multiplicación escogida obtiene el producto $A_i \times A_{i+1}$, para el cual las dimensiones son d_{i-1}, d_i y d_{i+1}. Sea $B = A_i \times A_{i+1}$; entonces las dimensiones de B son d_{i-1} por d_{i+1}. El subproblema restante consiste en multiplicar

$$\begin{array}{ccccccccc} A_1 & \times \cdots \times & A_{i-1} & \times & B & \times & A_{i+2} & \times \cdots \times & A_n \\ d_0 \times d_1 & & d_{i-2} \times d_{i-1} & & d_{i-1} \times d_{i+1} & & d_{i+1} \times d_{i+2} & & d_{n-1} \times d_n \end{array}$$

Supondremos que la sucesión de índices en sí se guarda en un arreglo `suc` cuyos índices comienzan en cero y que `longsuc` es la longitud de `suc`. El bosquejo del método es

```
mmIntento1(dim, longsuc, suc) // BOSQUEJO
    if (longsuc < 3)
        mejorCosto = 0; // caso base, un arreglo o ninguno.
    else
        mejorCosto = ∞;
        for (i = 1; i <= long−1; i++)
            c = costo de la multiplicación que está en la posición suc[i].
            nuevaSuc = suc sin el i-ésimo elemento.
            b = mmIntento1(dim, longsuc−1, nuevaSuc);
            mejorCosto = min(mejorCosto, b + c);
    return mejorCosto;
```

La ecuación de recurrencia para este algoritmo es

$$T(n) = (n - 1)T(n - 1) + n.$$

La solución está en $\Theta((n - 1)!)$, pero lo que buscamos es mejorar el desempeño del algoritmo recursivo convirtiéndolo en un algoritmo de programación dinámica.

Para diseñar una versión de programación dinámica, primero necesitamos analizar el grafo de subproblemas. ¿A cuántos subproblemas se puede llegar desde el problema inicial, descrito por la sucesión de índices $0, \ldots, n$? Aquí nos topamos con un obstáculo importante. Aunque al principio las subsucesiones son unos cuantos subintervalos continuos, se fragmentan cada vez más a medida que aumenta la profundidad del problema. Por ejemplo, con $n = 10$ después de escoger los operadores de multiplicación 1, 4, 6, 9, la subsucesión de índices restantes se convierte en 0, 2, 3, 5, 7, 8, 10. No hay forma concisa de especificar estas subsucesiones. Básicamente, toda subsucesión (de por lo menos tres elementos) de la sucesión original $(0, \ldots, n)$ es un subproblema alcanzable. Existen aproximadamente 2^n sucesiones así (véase el ejercicio 10.3), así que el número de subproblemas es exponencial. Este grafo simplemente es demasiado grande para efectuar en él una búsqueda eficiente.

Esto ilustra uno de los principios más importantes del diseño de un algoritmo de programación dinámica. Los subproblemas deben tener un *identificador conciso*. Esto limita el tamaño máximo del grafo de subproblemas (en términos de vértices; podría haber más aristas) y del diccionario al número de posibles identificadores (dentro de los intervalos que es preciso resolver). Recordemos que el *identificador*, o *id*, de un elemento lo identifica de manera única en el diccionario (sección 2.5.3). No puede haber en el diccionario más elementos que *identificadores* distintos haya. Por tanto, si nos concentramos en que el tamaño máximo del diccionario sea una función polinómica del tamaño de la entrada (y lo más pequeño posible) garantizaremos que la búsqueda primero en profundidad del grafo de subproblemas se podrá efectuar en tiempo polinómico.

Con base en estas consideraciones, nos damos cuenta de que necesitamos una idea distinta para descomponer el problema en subproblemas. Examinar el subproblema creado después de la primera multiplicación de matrices no funcionó. ¿Qué tal el subproblema que se crea al escoger la *última* multiplicación de matrices? Supóngase que la *última* multiplicación de matrices está en la posición *i*. Esto crea en realidad *dos* subproblemas:

1. Multiplicar $A_1, \ldots, A_i$ con índices de dimensión $0, \ldots, i$; es decir,

$$\begin{array}{ccccc} A_1 & \times & A_2 & \times \cdots \times & A_i \\ d_0 \times d1 & & d_1 \times d_2 & & d_{i-1} \times d_i \end{array} = \begin{array}{c} B_1 \\ d_0 \times d_i \end{array}$$

2. Multiplicar $A_{i+1}, \ldots, A_n$ con índices de dimensión $i, \ldots, n$.

$$\begin{array}{ccccc} A_{i+1} & \times & A_{i+2} & \times \cdots \times & A_n \\ d_i \times d_{i+1} & & d_{i+1} \times d_{i+2} & & d_{n-1} \times d_n \end{array} = \begin{array}{c} B_2 \\ d_i \times d_n \end{array}$$

El último paso es la multiplicación de B_1 por B_2, y su costo se basa en (d_0, d_i, d_n).

A primera vista no es obvio que esto sea mejor que nuestra primera estrategia. Sin embargo, observamos que cada subproblema se puede identificar (hasta ahora) con un par de enteros, $(0, i)$ e (i, n). Es decir, la sucesión de índices para el primer subproblema es $(0, 1, \ldots, i)$, pero dado que los elementos son contiguos sólo hay que dar los extremos. (Un par $(j - 1, j)$ representa a A_j solo y tiene costo 0.) Igual que antes, el arreglo de dimensiones, $d_0, \ldots, d_n$, no se modifica y podría ser un arreglo global.

Un examen más a fondo revela que si para un subproblema siempre se escoge como índice la última multiplicación, cada subproblema nuevo así creado se puede describir con un solo par de enteros. Por ejemplo, si para el subproblema (i, n) se escoge k, los subproblemas nuevos serán (i, k) y (k, n). Así, vemos que este método de descomposición de problemas sólo crea $\Theta(n^2)$ subproblemas distintos en el grafo de subproblemas.

Igual que antes, no sabemos qué opción escogida como última multiplicación producirá el costo total más bajo, así que es necesario evaluar todas las opciones. El objetivo de `mmIntento2(dim, bajo, alto)` es hallar el costo óptimo para el subproblema especificado por (`bajo`, `alto`), donde `bajo` < `alto`. El bosquejo es similar a `mmIntento1`:

```
mmIntento2(dim, bajo, alto)  // BOSQUEJO
 1. if (alto − bajo == 1)
 2.     mejorCosto = 0;  // Caso base: sólo una matriz.
 3. else
 4.     mejorCosto = ∞;
 5. for (k = bajo+1; k <= alto − 1; k ++)
 6.     a = mmIntento2(dim, bajo, k);
 7.     b = mmIntento2(dim, k, alto);
 8.     c = costo de la multiplicación de matrices de la
        posición k, con dimensiones dim[bajo], dim[k], dim[alto].
 9.     mejorCosto = min(mejorCosto, a + b + c);
10. return mejorCosto;
```

Este algoritmo, igual que `mmIntento1`, es de retroceso. La ecuación de recurrencia exacta para este algoritmo es complicada pero podemos obtener una versión simplificada que nos dice que el tiempo es mayor que 2^n (véase el ejercicio 10.4). Ya esperábamos esto, porque los algoritmos de retroceso suelen ser exponenciales, pero confiamos en poder mejorar el desempeño del algoritmo recursivo natural convirtiéndolo en un algoritmo de programación dinámica.

Una vez más, consideremos el grafo de subproblemas, donde el problema inicial está descrito por el par $(0, n)$. Los vértices (subproblemas) se identifican con un par de enteros, digamos $(i,$

j), dentro del intervalo 0, ..., n, con $i < j$, así que hay aproximadamente $n^2/2$ pares. Para el subproblema identificado por el par (i, j), hay dos subproblemas que resolver con invocaciones recursivas para cada k entre $i + 1$ y $j - 1$, lo que implica que menos de $2n$ aristas salen del vértice (i, j). En total, todo el grafo de subproblemas tiene menos de n^3 aristas, por lo que una búsqueda primero en profundidad lo puede recorrer en tiempo $O(n^3)$. Esto no es excesivo, así que podemos efectuar la conversión de `mmIntento2` a un algoritmo de programación dinámica `mmIntento2PD` insertando pruebas para consultar soluciones en lugar de recalcularlas para luego almacenarlas conforme se van hallando.

En el bosquejo de procedimiento que sigue, el diccionario se llama `costo` y el identificador de un elemento es un par de enteros. Para usar un TDA genérico, tendríamos que incorporar el identificador en una clase organizadora, pero en vez de ello supondremos que hemos adaptado esta interfaz de diccionario de modo que reciba los dos enteros, `bajo` y `alto`, como parámetros individuales. Seguiremos usando las operaciones de diccionario `crear`, `miembro`, `recuperar` y `almacenar`. Los números de línea corresponden a `mmIntento2`.

```
mmIntento2PD(dim, bajo, alto, costo)  // BOSQUEJO
 1. if (alto − bajo == 1)
 2.     mejorCosto = 0;  // Caso base: sólo una matriz.
 3. else
 4.     mejorCosto = ∞;
 5. for (k = bajo+1; k <= alto − 1; k ++)
6a.     if (miembro(bajo, k) == false)
6b.       a = mmIntento2PD(dim, bajo, k, costo);
6c.     else
6d.       a = recuperar(costo, bajo, k);

7a.     if (miembro(k, alto) == false)
7b.       b = mmIntento2DP(dim, k, alto, costo);
7c.     else
7d.       b = recuperar(costo, k, alto);

 8.     c = costo de la multiplicación de matrices de la
        posición k, con dimensiones dim[bajo], dim[k], dim[alto].
 9.     mejorCosto = min(mejorCosto, a + b + c);
10a. almacenar(costo, bajo, alto, mejorCosto);
10b. return mejorCosto;
```

Puesto que los subproblemas se identifican con un par de enteros dentro del intervalo 0, ..., n, el diccionario se puede implementar con un arreglo de $(n + 1) \times (n + 1)$ elementos. En `mmIntento2PD` almacenamos y recuperamos el costo óptimo de los subproblemas. En el algoritmo completo que sigue el arreglo `costo` se complementa con el arreglo `ultimo`, que contendrá la *opción* óptima de índice de multiplicación para el subproblema. Un elemento con costo ∞ denota un subproblema no resuelto. "Desabstraeremos" el diccionario y accederemos a los arreglos directamente.

Ya vimos que `mmIntento2` se puede convertir en `mmIntento2PD` con unos cuantos cambios mecánicos, que implementan la *memo-ización*. El resultado se parece mucho al esqueleto DFS: las líneas 6 y 7 detectan subproblemas no resueltos (vértices blancos, o no descubiertos) en un ciclo que recorre todos los subproblemas necesarios (todas las aristas a vértices adyacentes). Los

subproblemas resueltos (vértices negros) simplemente se consultan. En la línea 10 (procesamiento en orden posterior) el subproblema actual queda resuelto (el vértice actual se colorea de negro). Puesto que `mmIntento2PD` corresponde a la parte recursiva del esqueleto (es decir, `dfs`), para completar la implementación necesitamos una "envoltura" análoga a `barridoDFS` que inicialice el arreglo `costo` con ∞ y efectúe la invocación de nivel más alto de `mmIntento2PD`.

Una alternativa, partiendo de `mmIntento2`, es determinar por inspección un orden topológico inverso adecuado. Si podemos hacerlo, simplemente resolveremos problemas en ese orden, y conforme vaya siendo necesario resolver cada subproblemas, todos los subproblemas que son sus dependencias ya se habrán resuelto (véase la sección 7.4.6 y el ejemplo 7.14). Vemos que el subproblema (`bajo`, `alto`) depende de (tiene aristas de dependencia a) (`bajo`, `k`) y (`k`, `alto`), para `bajo` $< k <$ `alto`. ¿Podemos encontrar un orden simple que haga que todas las aristas apunten de un número topológico más alto a uno más bajo? Invitamos a los lectores a tratar de encontrar un orden semejante antes de seguir leyendo.

■ ■ ■

Por las aristas mencionadas en el párrafo anterior, vemos que reducir el segundo índice sin modificar el primero deberá conducir a un número topológico menor. Invirtiendo la lógica, incrementar el segundo índice sin modificar el primero deberá conducir a un número topológico más alto. De forma similar, reducir el primer índice sin modificar el segundo deberá conducir a un número topológico más alto. Hay varios esquemas que funcionan. Optemos por crear un ciclo **`for`** doble que abarque los subproblemas necesarios, y por variar el primer índice del ciclo exterior. Entonces tendrá que disminuir conforme del ciclo avanza. Asimismo el segundo índice, que varía en el ciclo interior, necesitará aumentar conforme el ciclo avanza. He aquí un bosquejo de esa estrategia:

```
ordenMatriz(n, costo, ultimo)  // BOSQUEJO
    for (bajo = n−1; bajo >= 1; bajo --)
       for (alto = bajo+1; alto <= n; alto ++)
          Calcular la solución del subproblema (bajo, alto) y almacenarla en
          costo[bajo][alto] y ultimo[bajo][alto].
    return costo[0][n];
```

El procedimiento para calcular la solución del subproblema (`bajo`, `alto`) es similar a `mmIntento2PD`, salvo que sabemos que las pruebas de las líneas 6a y 7a siempre darán **`false`**, por lo que sólo se necesitan las líneas 6d y 7d de esos enunciados compuestos.

El algoritmo final calcula las opciones óptimas y sus costos y los almacena en `ultimo` y `costo`, y luego invoca otra subrutina, `envoltExtraerOrden` (algoritmo 10.2) para extraer de `ultimo` la sucesión óptima de multiplicaciones real.

Algoritmo 10.1 Orden óptimo de multiplicación de matrices

Entradas: Un arreglo `dim` que contiene $d_0, \ldots, d_n$, las dimensiones de las matrices; n, el número de matrices a multiplicar.

Salidas: Un arreglo `ordenMult` en el que el i-ésimo elemento, para $1 \leq i \leq n - 1$, contiene el índice de la i-ésima multiplicación, dentro de una sucesión óptima. El arreglo se pasa como parámetro y el algoritmo lo llena. El algoritmo también devuelve el costo total del mejor orden de multiplicación.

Comentarios: El subproblema identificado por el par (`bajo`, `alto`) es el problema de optimar el cálculo de $A_{bajo+1} \times \cdots \times A_{alto}$. Por tanto, el problema de nivel más alto se especifica como (0, n). Este algoritmo usa dos arreglos bidimensionales, `costo` y `ultimo`, donde `ultimo` representa el índice de la última multiplicación que se hará en el subproblema. El costo se calcula en la subrutina `costoMult`, que se implementa de modo que devuelva el número de multiplicaciones necesarias, pero podría codificarse de modo que calcule cualquier función de costo que se desee.

```
float ordenMatrices(int[] dim, int n, int[]ordenMult)
     int[][] ultimo = new int[n+1][n+1];
     float[][] costo = new float[n+1][n+1];
     int bajo, alto, k, mejorUltimo;
     float mejorCosto;


     for (bajo = n-1; bajo >= 1; bajo --)
        for (alto = bajo+1; alto <= n; alto ++)
           // Calcular la solución del subproblema (bajo, alto) y
           // almacenarla en costo[bajo][alto] y ultimo[bajo][alto].
           if (alto - bajo == 1)
              mejorCosto = 0;
              mejorUltimo = -1;
           else
              mejorCosto = ∞;
           for (k = bajo+1; k <= alto-1; k ++)
              float a = costo[bajo][k];
              float b = costo[k][alto];
              float c = costoMult(dim[bajo], dim[k], dim[alto]);
              if (a + b + c < mejorCosto)
                 mejorCosto = a + b + c;
                 mejorUltimo = k;
              // Continúa el for(k)
           costo[bajo][alto] = mejorCosto;
           ultimo[bajo][alto] = mejorUltimo;


     envoltExtraerOrden(n, ultimo, ordenMult);
     return costo[0][n];


float costoMult(float Dizq, float Dmed, float Dder)
     return Dizq * Dmed * Dder;
```

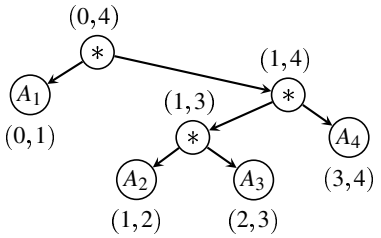


Figura 10.2 Árbol de expresión aritmética que corresponde a la solución del ejemplo 10.5. Cada nodo se identifica con un subproblema y representa una matriz o bien una multiplicación a efectuar.

Ejemplo 10.5 Matrices para `costo` y `ultimo`

Para la sucesión de matrices del ejemplo 10.4, $d_0 = 30$, $c_1 = 1$, $d_2 = 40$, $d_3 = 10$ y $d_4 = 25$. El algoritmo `ordenMatrices` produciría las tablas `costo` y `ultimo` siguientes. Los elementos "·" no se calcularon. Los elementos para los que `ultimo` es -1 y `costo` es 0 son subproblemas que consisten en una sola matriz.

$$\texttt{costo} = \begin{bmatrix} \cdot & 0 & 1200 & 700 & 1400 \\ \cdot & \cdot & 0 & 400 & 650 \\ \cdot & \cdot & \cdot & 0 & 10000 \\ \cdot & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \qquad \texttt{ultimo} = \begin{bmatrix} \cdot & -1 & 1 & 1 & 1 \\ \cdot & \cdot & -1 & 2 & 3 \\ \cdot & \cdot & \cdot & -1 & 3 \\ \cdot & \cdot & \cdot & \cdot & -1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

El costo de la mejor manera de multiplicar las matrices es `costo[0][4]`, que es 1400. Más adelante veremos cómo se extrae de la tabla el mejor orden de multiplicación. ■

Obsérvese que escoger la *última* multiplicación a efectuar equivale a escoger la raíz de un árbol de expresión aritmética para las multiplicaciones; cada nodo interno denota una multiplicación de matrices y las hojas son las matrices. La figura 10.2 muestra un ejemplo para la solución del ejemplo 10.5. En `mmIntento2PD` el primer subproblema decide recursivamente cuál es el mejor subárbol izquierdo y el segundo subproblema decide recursivamente cuál es el mejor subárbol derecho; el orden de `ordenMatrices` no deja esto muy en claro. Un recorrido en orden posterior del árbol enumera las multiplicaciones en el orden en el que la evaluación estándar de expresiones las ejecuta. Esto se hace en el algoritmo 10.2.

Análisis de `mmIntento2PD` y `ordenMatrices`

No hemos examinado todavía la subrutina `extraerOrden`, pero su costo es pequeño en comparación con el del algoritmo principal. Tenemos dos versiones, `mmIntento2PD` y `ordenMatrices`, que efectúan aproximadamente el mismo trabajo pero en diferente orden. En el caso de `mmIntento2PD`, simplemente observamos que es en esencia una búsqueda primero en profundidad en un grafo de $\Theta(n^2)$ vértices y $\Theta(n^3)$ aristas, con una cantidad constante de procesamiento por arista y por vértice. En el caso de `ordenMatrices`, el cuerpo del ciclo **`for`** más interior requiere tiempo constante y se ejecuta $\Theta(n^3)$ veces. Por tanto, ambas implementaciones se ejecutan en tiempo $\Theta(n^3)$. Esto es mucho mejor que ejecutar un número exponencial de pasos.

El espacio extra que se requiere para los arreglos bidimensionales `costo` y `ultimo` está en $\Theta(n^2)$, es decir, cuadrático en términos del tamaño de las entradas y de las salidas, estando ambos en $\Theta(n)$. Una solución recursiva usaría únicamente espacio en $\Theta(n)$ (para la pila de marcos de activación). El gasto de espacio extra para producir un algoritmo mucho más rápido bien vale la pena.

Existe un algoritmo $\Theta(n^2)$ para determinar el mejor orden de multiplicación de una sucesión de matrices. Sin embargo, es especializado para la función de costo dada por `costoMult` en el algoritmo 10.1, mientras que el algoritmo 10.1 no depende de ninguna función de costo específica. Véanse las notas y referencias al final del capítulo.

10.3.4 Extracción del orden óptimo

El procedimiento recursivo siguiente, `extraerOrden`, extrae de la tabla `ultimo` calculada por `ordenMatrices` un orden óptimo para multiplicar las matrices. Se le invoca desde su envoltura, `envoltExtraerOrden`, que se invoca desde `ordenMatrices` como último paso de su cálculo.

Primero, `envoltExtraerOrden` inicializa una variable global, `ordenMultSiguiente`, que es el índice para llenar el arreglo de salida `ordenMult`. Luego se invoca `extraerOrden` para que haga el trabajo. El objetivo de `extraerOrden(bajo, alto, ultimo, ordenMult)` es llenar el arreglo `ordenMult` con el orden de multiplicación óptimo para el subproblema especificado por (`bajo`, `alto`).

El algoritmo podría reconocerse como un recorrido en orden posterior de un árbol binario que está definido implícitamente en `ultimo` como sigue. Sea `k = ultimo[bajo][alto]` (cuando `alto − bajo` $>$ 1). El nodo de árbol (`bajo`, `alto`) tiene como hijos izquierdo y derecho los nodos (`bajo`, `k`) y (`k`, `alto`), respectivamente. Cuando `alto − bajo` $= 1$, el nodo es una hoja. Obsérvese que `extraerOrden` es recursivo. ¿Por qué es apropiada aquí la recursión, en vez de la programación dinámica?

Algoritmo 10.2 Extracción del orden de multiplicación óptimo

Entradas: El número de matrices, n; la matriz `ultimo`, calculada por `ordenMatrices` en el algoritmo 10.1.

Salidas: El arreglo `ordenMult` descrito en el algoritmo 10.1. El arreglo se pasa como parámetro y este procedimiento llena las posiciones 1 a $n - 1$.

```
int ordenMultSiguiente;

envoltExtraerOrden(n, ultimo, ordenMult);
    ordenMultSiguiente = 0;
    extraerOrden(0, n, ultimo, ordenMult);

extraerOrden(bajo, alto, ultimo, ordenMult)
    int k;
    if (alto − bajo > 1)
       k = ultimo[bajo][alto];
       extraerOrden(bajo, k, ultimo, ordenMult);
       extraerOrden(k, alto, ultimo, ordenMult);
       ordenMult[ordenMultSiguiente] = k;
       ordenMultSiguiente ++;
```

Ejemplo 10.6 Extracción del orden de multiplicación

En la figura 10.2 se muestra el árbol implícito que `extraerOrden` recorre para la tabla `ultimo` del ejemplo 10.5. El orden posterior de los nodos es (0, 1), (1, 2), (2, 3), (1, 3), (3, 4), (1, 4), (0, 4). Sólo los nodos internos hacen que se escriba un elemento en el arreglo `ordenMult`; éstos son (1, 3), (1, 4) y (0, 4). Así pues

```
ordenMult[1] = ultimo[1][3] = 2
ordenMult[2] = ultimo[1][4] = 3
ordenMult[3] = ultimo[0][4] = 1
```

es decir, el orden óptimo determinado por el algoritmo es 2, 3, 1, que corresponde a la factorización óptima $A_1((A_2A_3)A_4)$ que se muestra en el ejercicio 10.4. ■

Análisis de `extraerOrden`

Cada invocación de `extraerOrden` (algoritmo 10.2) visita un nuevo nodo del árbol de expresión, que tiene $2n - 1$ nodos, así que hay $2n - 1$ invocaciones, y `extraerOrden` tarda un tiempo $\Theta(n)$. El árbol podría tener una profundidad en $\Theta(n)$, en cuyo caso la pila de marcos de activación requeriría un espacio $\Theta(n)$. Sin embargo, dado que la estructura de datos que se está procesando ocupa un espacio $\Theta(n^2)$, las necesidades de espacio de este algoritmo probablemente serán insignificantes.

10.4 Construcción de árboles de búsqueda binaria óptimos

En esta sección consideraremos el problema de hallar la mejor forma de acomodar un conjunto de claves (de algún conjunto ordenado linealmente) en un árbol de búsqueda binaria a modo de reducir al mínimo el tiempo de búsqueda medio si sabemos que algunas claves se consultan con mayor frecuencia que otras. En un árbol de búsqueda binaria las claves que están en los nodos satisfacen la propiedad de árbol de búsqueda binaria dada en la definición 6.3. Recordemos que un recorrido en orden interno de un árbol de búsqueda binaria visita los nodos en orden creciente según sus claves. En la figura 10.3 se da un ejemplo. Podría ser recomendable que el lector repase el algoritmo 6.1 para recuperación en un árbol de búsqueda binaria antes de continuar.

Utilizaremos como medida del trabajo el número de comparaciones de claves efectuadas, o el número de nodos del árbol examinados, durante la búsqueda de una clave. Supondremos, co-

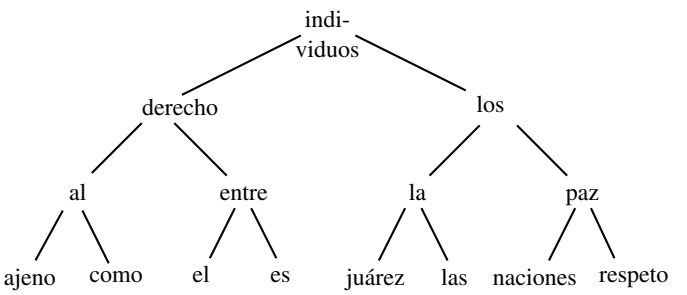


Figura 10.3 Un árbol de búsqueda binaria

mo hicimos en la sección 6.4, que podemos efectuar comparaciones de tres vías, así que el número de comparaciones efectuadas para hallar una clave en el árbol es 1 más la profundidad del nodo que contiene la clave.

Supóngase ahora que las claves son $K_1, K_2, \ldots, K_n$ y que la probabilidad de que se busque cada clave es $p_1, p_2, \ldots, p_n$, respectivamente. Las probabilidades normalmente se obtienen de la experiencia previa o de alguna otra cosa que se sepa acerca de la aplicación. Se pueden usar frecuencias (que no tienen que sumar 1) en lugar de probabilidades.

Supóngase que hemos acomodado las claves en un árbol de búsqueda binaria T. Sea c_i el número de comparaciones que el algoritmo 6.1 efectúa para hallar K_1 (es decir, la profundidad de K_i más 1). El número de nodos de T que se examinan en promedio es

$$A(T) = \sum_{i=1}^{n} p_i c_i. \tag{10.1}$$

Si todas las claves tienen la misma probabilidad de ser buscadas ($p_i = 1/n$ para toda i), lo mejor es mantener el árbol lo más equilibrado que se pueda; el número medio de comparaciones es aproximadamente lg n (ejercicio 10.8), que es prácticamente igual al peor caso si el árbol está equilibrado. En cambio, si es mucho más probable que se busquen unas claves que otras, una estructura desequilibrada podría hacer que se efectuaran menos comparaciones en promedio.

Ejemplo 10.7 Cálculo del tiempo de búsqueda medio

La tabla 10.1 muestra una lista de claves y datos acerca del número de veces que cada clave se buscó en un experimento (hipotético). La probabilidad de cada clave se calcula a partir de los datos. (Los datos se escogieron a modo de facilitar el cálculo; no son demasiado realistas.) Supón-

Clave	Número de búsquedas	Probabilidad (p_i)
ajeno	30	0.150
al	5	0.025
como	10	0.050
derecho	25	0.025
el	10	0.050
entre	30	0.125
es	10	0.025
individuos	15	0.075
juárez	5	0.075
la	5	0.050
las	30	0.150
los	15	0.075
naciones	15	0.050
paz	5	0.025
respeto	10	0.050
	Total = 200	Total = 1.000

Tabla 10.1 Datos de las claves

Clave	Probabilidad (p_i)	Comparaciones (c_i)	$p_i\, c_i$
y	0.150	4	0.600
ajeno	0.025	4	0.100
al	0.050	3	0.150
como	0.125	4	0.500
derecho	0.050	2	0.100
el	0.150	4	0.600
entre	0.050	3	0.150
es	0.075	4	0.300
individuos	0.025	1	0.025
juárez	0.025	4	0.100
la	0.150	3	0.450
las	0.075	4	0.300
los	0.075	2	0.150
naciones	0.025	2	0.050
paz	0.050	3	0.150
respeto	0.050	4	0.200
			Total = 3.325

Tabla 10.2 Cálculo del tiempo medio de búsqueda

gase ahora que se ha construido un árbol de búsqueda binaria como el de la figura 10.3. La tabla 10.2 muestra el cálculo del tiempo medio de búsqueda.

El tiempo medio de búsqueda es de 3.325. Deberá ser obvio que este árbol no es óptimo. Las dos claves que se buscan con mayor frecuencia, *el* y *la*, están en el último y el penúltimo niveles, respectivamente, por lo que requieren un tiempo de búsqueda largo. Es posible que colocar *el* en la raíz no mejore el promedio porque, para mantener la propiedad de árbol de búsqueda binaria, la mayor parte de las claves quedaría en el subárbol derecho. No es evidente que el panorama mejore si se coloca *la* en la raíz. No obstante, ataquemos el problema de manera sistemática. ■

Queremos hallar un árbol de búsqueda binaria para las claves $K_1, K_2, \ldots, K_n$ cuyas probabilidades de ser buscadas son $p_1, p_2, \ldots, p_n$, de modo que el tiempo medio de búsqueda sea mínimo. Supondremos que las claves ya están ordenadas. Si escogemos K_k como raíz del árbol, $K_1, \ldots, K_{k-1}$ deberán colocarse en el subárbol izquierdo, y $K_{k+1}, \ldots, K_n$, en el derecho. Ahora necesitamos acomodos óptimos para los dos subárboles. Véase la figura 10.4. Puesto que no sabemos cuál clave sería mejor como raíz, determinaremos el mínimo de todas las opciones.

El plan anterior se parece mucho a la forma en que se descompuso el problema del orden de multiplicación de matrices. Esto sugiere identificar cada subproblema con el par (`bajo`, `alto`), los índices bajo y alto del subintervalo de claves representado por ese subproblema. Así, el subproblema (`bajo`, `alto`) consiste en hallar el árbol de búsqueda binaria con el más bajo costo de recuperación *ponderado* para las claves $K_{bajo}, \ldots, K_{alto}$ con pesos $p_{bajo}, \ldots, p_{alto}$. Estamos cambiando nuestra terminología de probabilidades a pesos porque, en los subproblemas, las p no suman 1.

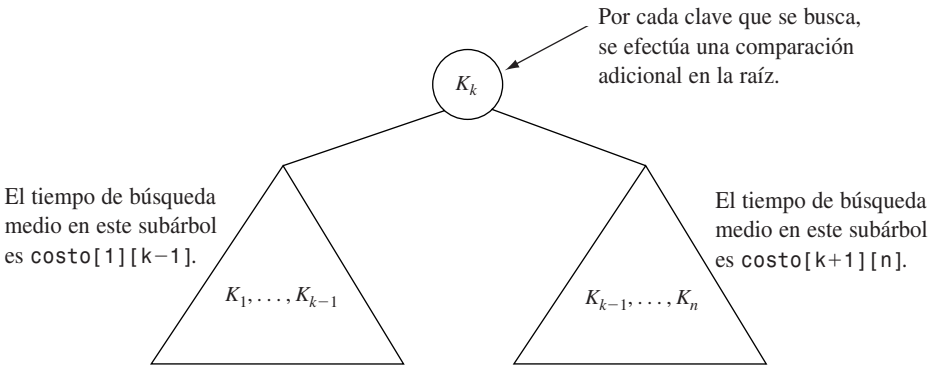


Figura 10.4 Se escoge K_k como raíz

Definición 10.3

Adoptamos la notación siguiente:

1. Definimos A(bajo, alto, r) como el costo ponderado mínimo del subproblema (bajo, alto) cuando se escoge K_r como raíz de su árbol de búsqueda binaria.
2. Definimos A(bajo, alto) como el costo ponderado mínimo del subproblema (bajo, alto), considerando todas las raíces que podrían escogerse.
3. Definimos p(bajo, alto) $= p_{bajo} + \cdots + p_{alto}$; es decir, la probabilidad de que la clave buscada sea *alguna* clave dentro del intervalo $K_{bajo}, \ldots, K_{alto}$. Llamaremos a esta probabilidad el *peso del subproblema* (bajo, alto). ■

Si el costo de recuperación ponderado para un árbol dado que contiene $K_{bajo}, \ldots, K_{alto}$ es W (suponiendo que es todo el árbol, así que la profundidad de su raíz es cero) entonces, si la raíz del subárbol está en la profundidad 1, el costo de recuperación ponderado es ($W + p$(bajo, alto)). Es decir, toda búsqueda que entre en este subárbol requerirá una comparación más que si el subárbol fuera el árbol completo, la probabilidad de que la búsqueda entre en este subárbol es sólo p(bajo, alto). (Véase la figura 10.4.) Esta relación nos permite combinar soluciones recursivas de subproblemas para obtener la solución del problema mayor.

$$\begin{aligned} A(\text{bajo}, \text{alto}, r) &= pr + p(\text{bajo}, r-1) + A(\text{bajo}, r-1) \\ &\quad + p(r+1, \text{alto}) + A(r+1, \text{alto}) \\ &= p(\text{bajo}, \text{alto}) + A(\text{bajo}, r-1) + A(r+1, \text{alto}), \qquad (10.2) \end{aligned}$$

$$A(\text{bajo}, \text{alto}) = \min \{A(\text{bajo}, \text{alto}, r) \mid \text{bajo} \leq r \leq \text{alto}\}. \qquad (10.3)$$

Podríamos escribir un procedimiento recursivo para calcular A(bajo, alto) con base en las ecuaciones (10.2) y (10.3). Sin embargo, al igual que en el problema del orden de la multiplicación de matrices que estudiamos en la sección 10.3, observaríamos que una solución recursiva efectúa mucho trabajo repetido. El tiempo de ejecución del algoritmo sería exponencial. Una vez más, pa-

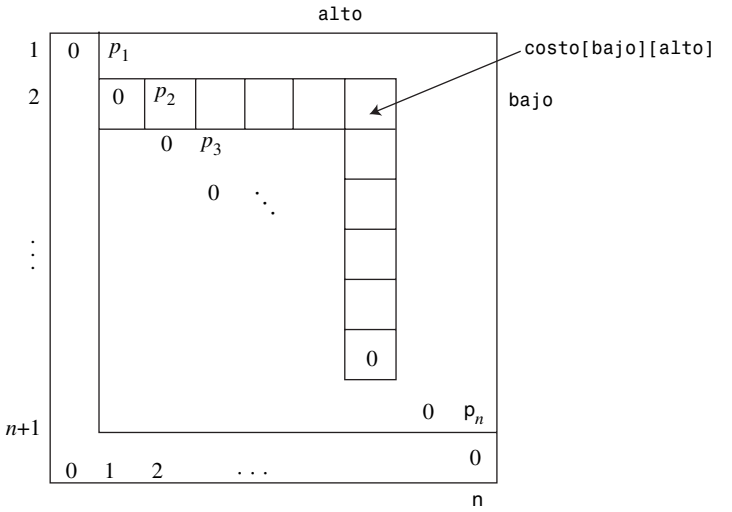


Figura 10.5 Cálculo de `costo[bajo][alto]`

ra evitar el trabajo repetido, definimos un diccionario, que se implementa con dos arreglos bidimensionales de tamaño $(n + 2) \times (n + 1)$, llamados `costo` y `raiz`.

Al igual que en el problema del orden de multiplicación de matrices, los subproblemas de los que `costo[bajo][alto]` depende están en una fila de número más alto (primer índice) o en una columna de número más bajo (segundo índice). Véase la figura 10.5. En vez de seguir el orden que seguiría el procedimiento recursivo, podemos calcularlos en un ciclo doble que opera hacia atrás según el primer índice y hacia adelante según el segundo, como hicimos en el algoritmo 10.1.

Algoritmo 10.3 Árbol de búsqueda binaria óptimo

Entradas: Un arreglo `prob` que contiene las probabilidades $p_1, \ldots, p_n$ de cada clave; n, el número de claves.

Salidas: Los arreglos bidimensionales `costo` y `raiz`, para los que se ha reservado espacio para $(n + 2) \times (n + 1)$ elementos, contando a partir del cero. Los arreglos se pasan como parámetros y el algoritmo los llena. El primer índice 0 no se usa. Para el subintervalo de claves $K_{bajo}, \ldots, K_{alto}$, donde $1 \leq$ `bajo` $\leq$ `alto` $\leq n$, `costo[bajo][alto]` da el costo de búsqueda ponderado más bajo y `raiz[bajo][alto]` da la clave que sería la mejor opción como raíz para el árbol de búsqueda binaria que incluye este subintervalo de claves. El costo óptimo para todo el árbol está en `costo[1][n]`.

Comentarios: Un par $(i, i - 1)$ representa un árbol vacío, cuyo costo es cero. El arreglo `costo` tiene una fila adicional (cuyo índice es $n + 1$) para simplificar las condiciones de frontera. La fila extra sólo se usa para almacenar el árbol vacío $(n + 1, n)$. Cabe señalar que $p(i, j) = p_i + \cdots + p_j$, igual que en el texto, y que $p(i, i - 1) = 0$.

```
BSToptimo(prob, n, costo, raiz) // BOSQUEJO
     for (bajo = n + 1; bajo >= 1; bajo --)
          for (alto = bajo−1; alto <= n; alto ++)
             mejorOpcion(prob, costo, raiz, bajo, alto);
     return costo;

/** Calcular la solución del subproblema (bajo, alto) */
mejorOpcion(prob, costo, raiz, bajo, alto) // BOSQUEJO
     if (alto < bajo)
          mejorCosto = 0; // árbol vacío
          mejorRaiz = −1;
     else
          mejorCosto = ∞;
     for (r = bajo; r <= alto; r ++)
          rCosto = p(bajo, alto) + costo[bajo][r−1] + costo[r+1][alto];
          if (rCosto < mejorCosto)
             mejorCosto = rCosto;
             mejorRaiz = r;
          // Continuar el ciclo
     costo[bajo][alto] = mejorCosto;
     raiz[bajo][alto] = mejorRaiz;
     return;
```

Una función recursiva para construir (y devolver) el árbol de búsqueda binaria óptimo, utilizando el TDA `ArbolBin` de la sección 2.3.3, sería similar al algoritmo 10.2 (véase el ejercicio 10.10).

Análisis

Gran parte del análisis es similar al del algoritmo 10.1. La función $p(i, j) = p_i + \cdots + p_j$ no se tiene que calcular "desde cero" en cada ocasión. Dejamos como ejercicio idear una forma eficiente de calcular estas sumas (ejercicio 10.11). También, si el cálculo con enteros es más rápido o recomendable por cualquier razón, se podrían usar directamente los datos de búsquedas anteriores de las claves (la segunda columna de la tabla 10.1, por ejemplo) en lugar de probabilidades como pesos para las claves. En todo caso, la cantidad de trabajo que el algoritmo 10.3 efectúa obviamente está en $\Theta(n^3)$.

10.5 División de sucesiones de palabras en líneas

En esta sección abordaremos el problema de separar una sucesión de palabras para formar una serie de líneas que constituyen un párrafo. El objetivo es evitar demasiados espacios extra en cualquier línea. Éste es un problema importante en tipografía computarizada. Dado que no importa si quedan espacios extra en la última línea del párrafo, es natural escoger el párrafo como la unidad que se optimará. Desde luego, es preciso mantener el orden de las palabras al colocarlas en líneas. La optimización de la división en líneas, como suele llamarse este problema, se intro-

dujo en el sistema de tipografía T_EX (se pronuncia "tech") inventado por Don Knuth y sus estudiantes en la Stanford University. La versión que estudiaremos está muy simplificada: supondremos que todas las letras y espacios tienen la misma anchura.

Las entradas del problema de división de líneas son una sucesión de n longitudes de palabra, $w_1, \ldots, w_n$, que representan la longitud de las palabras que constituyen un párrafo y una anchura de línea W. Para simplificar los cálculos, supondremos que cada w_i incluye en su cuenta un espacio al final de la palabra (por ejemplo, w sería 4 para "que") y que W incluye un espacio extra al final de la línea (o sea que, si queremos líneas con anchura real de 80, especificaremos $W = 81$).

La restricción básica de la colocación de palabras es que, si las palabras i-ésima a j-ésima se colocan en una sola línea, $w_i + \cdots + w_j \leq W$. En este caso el número de espacios extra es

$$X = W - (w_i + \cdots + w_j).$$

Suponemos que el castigo por incluir espacios extra es alguna función de X. En nuestra explicación el castigo por línea será X^3, pero el método de resolución deberá funcionar con diversas funciones de castigo. (Un castigo más realista podría depender también del número de palabras que hay en la línea, porque el espacio extra se puede repartir entre las palabras.) No hay castigo por espacios extra en la última línea del párrafo. El castigo del párrafo es la suma de los castigos de las líneas individuales y se debe reducir al mínimo.

Existe un algoritmo codicioso sencillo para este problema: simplemente se colocan en la primera línea tantas palabras como quepan en ella, luego en la segunda línea, y así sucesivamente hasta terminar el párrafo. Aunque esto no garantiza una división óptima de las líneas (sugerimos al lector inventar un contraejemplo), en la práctica funciona "muy bien" en la generalidad de los casos, en consecuencia es el método que usan muchos paquetes de software.

Ejemplo 10.8 División de líneas

Consideremos la cita que aparece al principio del capítulo, que tomaremos como todo un párrafo:

i	1	2	3	4	5	6	7	8	9	10
	Quienes	no	pueden	recordar	el	pasado	están	condenados	a	repetirlo.
w_i	6	4	7	9	4	5	4	10	3	7

Supóngase $W = 17$. La estrategia codiciosa agrupa las palabras en líneas como sigue:

palabras	(1, 2)	(3, 4)	(5, 6, 7)	(8, 9)	(10)
X	6	1	1	4	0
castigo	216	1	1	64	0

¿Es esto óptimo? ■

Ataquemos el problema descomponiéndolo en subproblemas, como hicimos con los problemas de secciones anteriores de este capítulo. Supóngase que dividimos en líneas las palabras $1, \ldots, k$ y luego dividimos en líneas las palabras $k + 1, \ldots, n$, de forma independiente. Si resolvemos cada subproblema de manera óptima, ¿es óptima su combinación? No necesariamente, porque k podría no ser un buen lugar para un salto de línea. Por otra parte, si retrocedemos exa-

minando todas las posibles k, una de ellas será óptima. Esto se parece a los enfoques que adoptamos con los problemas del orden de multiplicación de matrices y del árbol de búsqueda binaria óptimo. Un subproblema se identifica con un par de índices (i, j). El objetivo del subproblema es dividir las palabras i-ésima a j-ésima en líneas incurriendo en un castigo mínimo.

¿El plan anterior da una solución recursiva correcta? Hay que tener cuidado, porque el castigo en la última línea del párrafo es cero, pero el castigo en la última línea de todos los subproblemas que *no* terminan el párrafo se calcula de la forma acostumbrada. Así que en realidad tenemos dos clases de subproblema. A sabiendas de esto, podríamos tratar de demostrar que, escogiendo alguna k, la combinación de las soluciones óptimas para los subintervalos $(1, k)$ y $(k + 1, n)$ da una solución óptima para $(1, n)$.

Antes de entrar en demostraciones, consideremos el tamaño del diccionario y del grafo de subproblemas. Hay aproximadamente $n^2/2$ subproblemas (vértices) y cada subproblema (i, j) tiene una arista a cerca de $j - i$ subproblemas distintos. En total, habrá $\Theta(n^3)$ aristas. Es indudable que un procedimiento recursivo de retroceso simple se ejecutará en tiempo exponencial, mientras que la versión de programación dinámica lo hará en tiempo polinómico, como vimos en los problemas anteriores de este capítulo. Sin embargo, ¿es ésta la mejor cota que podemos lograr? Si podemos idear una forma de identificar subproblemas *más concisa* que un par de enteros, el diccionario será más pequeño, pero también lo será el grafo de subproblemas. Sugerimos al lector tratar de hallar una estrategia de descomposición de problemas distinta que de pie a identificadores más concisos y a un diccionario más pequeño, antes de seguir leyendo.

■ ■ ■

La observación que hicimos de que hay dos clases de subproblemas en la estrategia propuesta nos da una pista para encontrar la solución. Una de esas clases incluye la última línea del párrafo; la otra no. Sin embargo, la primera clase en realidad se identifica con un *solo* entero, el primer índice de su subintervalo. Si ahí termina el párrafo, el final del subintervalo deberá ser n. Es decir, el identificador debe tener la forma (k, n), donde $1 \leq k \leq n$, pero n forma parte de las entradas y no cambia durante todo el problema. Por tanto, n no es una parte necesaria del identificador de diccionario en el caso de estos subproblemas. De hecho, sólo tenemos aproximadamente n subproblemas de la forma (k, n).

¿Realmente necesitamos los subproblemas de la forma (i, j), donde $j \neq n$? Apliquemos el método 99 (sección 3.2.2). Supóngase que *ya tenemos* una subrutina capaz de hallar soluciones óptimas de división de líneas para problemas de 99 palabras o menos —llamémosla `divLineas99`— y que esa subrutina supone que el final del problema es el final del párrafo. ¿Cómo podemos aprovechar esto para resolver el problema de división de líneas para $n = 100$ palabras o menos, es decir, para escribir `divLineas100`? Si lo meditamos un poco veremos que podemos efectuar iteraciones con base en el número de palabras que colocaremos en la *primera* línea. Si ese número es k, el problema restante consiste en colocar de manera óptima las palabras $k + 1, \ldots, n$ en el resto de las líneas. Sin embargo, ¡podemos hacerlo con `divLineas99`! Escogemos la k que reduce al mínimo el castigo combinado para la primera línea y las líneas restantes. Ahora desechamos los sufijos "100" y "99" para tener un procedimiento recursivo.

El algoritmo sigue siendo de retroceso, porque no sabemos cuál es la mejor k, así que hay que retroceder para examinar todas las opciones. Sin embargo, sólo necesitamos las opciones que acomodan todas las palabras escogidas en una línea, así que no hay más de $W/2$ opciones para k. (Recordemos que W es la anchura de línea y que hay por lo menos un espacio después de cada

palabra.) A continuación delineamos el procedimiento de retroceso. Es fácil demostrar por inducción que es correcto.

```
divLineas(w, W, i, n, L)   //BOSQUEJO
    if (w_i + ··· w_n <= W)
        Colocar todas las palabras en la línea L y asignar 0 a castigo.
    else
        Asignar a castigo el mínimo de kCastigo para todas las k > 0 tales que w_i +
        ··· + w_{i+k-1} ≤ W, donde X = W − (w_i + ··· + w_{i+k-1}) y kCastigo =
        costoLinea(X) + divLineas(w, W, i+k, n, L+1);
        Sea k_mín la k que produjo el castigo mínimo.
        Colocar las palabras de la i a la i + k_mín − 1 en la línea L.
    return castigo;
```

El identificador para el diccionario que se usa con `divLineas` es un solo entero dentro del intervalo 1, …, n. Por tanto, el diccionario puede ser un arreglo simple. La conversión de `divLineas` en `divLineasPD` sigue el método descrito en la definición 10.2:

1. Antes de efectuar una invocación recursiva, ver si la solución ya está en el diccionario;
2. Antes de regresar del procedimiento, *almacenar* la solución recién calculada.

Análisis

El grafo de subproblemas tiene cerca de n vértices (subproblemas) y cuando más $W/2$ aristas en cualquier vértice, así que el tiempo de ejecución está en $\Theta(Wn)$. El espacio de almacenamiento extra que se usa para el diccionario es $\Theta(n)$. Normalmente se considera que W es una constante, así que el tiempo de ejecución está en $\Theta(n)$. Al reducir el tamaño del diccionario de n^2 a n ahorramos *dos* grados en el polinomio del tiempo de ejecución: de $\Theta(n^3)$ a $\Theta(n)$.

10.6 Desarrollo de un algoritmo de programación dinámica

La esencia de los algoritmos de programación dinámica es que cambian espacio por rapidez almacenando soluciones a subproblemas en lugar de volver a resolverlos. Después de haber visto ejemplos, podemos hacer algunos comentarios generales acerca de cómo desarrollar una solución de programación dinámica para un problema.

1. Por lo regular es útil atacar el problema "de arriba hacia abajo" como si fuéramos a desarrollar un algoritmo recursivo; determinamos cómo resolver un problema grande suponiendo que conocemos soluciones para problemas más pequeños.
2. Si es evidente que guardando resultados de problemas más pequeños podremos evitar cálculos repetidos, definimos el diccionario apropiado para guardar los resultados y caracterizamos claramente las entradas del diccionario. Hay que procurar que el identificador de las entradas del diccionario sea lo más conciso posible; así, el diccionario y el número de subproblemas serán pequeños. Por ejemplo, en la sección 10.5 vimos que una estrategia de descomposición de problemas requería dos enteros para especificar un subproblema y daba pie a un algoritmo $\Theta(n^3)$, mientras que una estrategia distinta sólo requería un entero para especificar un subproblema y daba pie a un algoritmo $\Theta(n)$. Efectúese la conversión descrita en la definición 10.2. En ese momento se podrá determinar la inicialización apropiada.

3. Con base en el número de subproblemas (el tamaño máximo del diccionario es una cota superior) y el número de aristas del grafo de subproblemas, se puede analizar la complejidad del procedimiento de programación dinámica por su relación con la búsqueda primero en profundidad en el grafo de subproblemas.
4. Decidimos qué estructura de datos sería apropiada para el diccionario. En muchos casos lo mejor es un arreglo uni o bidimensional. En esos casos sencillos es posible eliminar la "abstracción" simplificando. Si necesitamos un diccionario más complejo (por ejemplo, porque los identificadores son demasiado ralos para que un arreglo simple resulte práctico), lo mejor podría ser ajustarnos a la división de tareas impuesta por la metodología de TDA.
5. Si es posible, analizamos la estructura del grafo de subproblemas e ideamos un orden más sencillo para calcular las entradas del diccionario. El requisito es que se calculen en *algún* orden topológico inverso. Entonces, todos los subproblemas de los que depende el subproblema actual ya se habrán calculado previamente.
6. Determinamos cómo obtener la solución del problema a partir de los datos del diccionario. En el caso de problemas como los de las secciones 10.3, 10.4 y 10.5, el costo óptimo está en un lugar específico del diccionario. El diccionario podría servir como entrada de otro algoritmo que extrae las opciones que dieron pie al costo óptimo. Vimos ejemplos en los que se extrajo el orden óptimo en que se deben multiplicar matrices o en que se construyó el árbol de búsqueda binaria óptimo. Puesto que el diccionario tiene datos para todos los subproblemas del grafo de subproblemas, por lo regular sólo un subconjunto pequeño de los datos tiene que ver con la solución óptima final.

La experiencia con la programación dinámica (y la recursión) ayuda a desarrollar una buena intuición respecto a qué funcionará mejor para diversos problemas. Algunos problemas que presentamos en capítulos anteriores también se pueden resolver dentro del marco de la programación dinámica, como el del conjunto independiente máximo en un árbol (ejercicio 3.13) y la suma de subsucesión máxima (ejercicio 4.59). Otros aparecerán en los capítulos 11 y 13.

Ejercicios

Sección 10.1 Introducción

★ **10.1** Defínase A_n como el número de marcos de activación que se crean durante el cálculo de F_n, el n-ésimo número de Fibonacci, utilizando la función recursiva natural `fib(n)` dada en el ejemplo 3.1. Observe que $A_0 = 1$, $A_1 = 1$ y $A_2 = 3$. Contando nodos en la figura 7.13, vemos que $A_6 = 25$.

★ **a.** Sea $\phi = \frac{1}{2}(5 + 1) \approx 1.618$. Ésta es la Razón Dorada. Demuestre que F_n está en $\Theta(\phi^n)$.

b. Demuestre que $A_n = 2F_{n+1} - 1$ para $n \geq 1$. En combinación con la parte (a), esto establece el orden asintótico del procedimiento `fib(n)`. Resulta interesante que la complejidad de `fib(n)` en términos de tiempo es $\Theta(F_n)$, y F_n es el valor que calcula.

10.2 Modifique el procedimiento del ejemplo 10.1, que calcula números de Fibonacci, de modo que sólo use un número constante de enteros como espacio de trabajo y pese a ello calcule F_n en tiempo $\Theta(n)$.

Sección 10.3 Multiplicación de una sucesión de matrices

10.3 Este ejercicio cuenta subsucesiones de la sucesión $(0, 1, \ldots, n)$. (Una subsucesión es cualquier subconjunto de los elementos de la sucesión en el mismo orden; no tienen que estar contiguos en la sucesión original.)

a. Demuestre que hay 2^{n+1} subsucesiones distintas, incluida la subsucesión vacía.
b. Demuestre que hay menos de $n^3/4$ subsucesiones con longitud 3 o menos.
c. Concluya que hay por lo menos 2^n subsucesiones distintas con longitud 4 o mayor, para $n \geq 5$.

10.4 El procedimiento de retroceso recursivo `mmIntento2` de la sección 10.3 calcula el costo óptimo de multiplicar una sucesión de matrices. Demuestre que el número de invocaciones recursivas que se efectúan durante la ejecución de `mmIntento2` está acotado por abajo por una función exponencial de n. (Un argumento similar demostraría que la solución recursiva correspondiente para el problema del árbol de búsqueda binaria óptimo de la sección 10.4 también es exponencial.)

Sugerencia: La ecuación de recurrencia exacta para `mmIntento2` es complicada pero podemos obtener una versión simplificada haciendo caso omiso de todos los subproblemas con excepción de los dos más grandes, que tienen tamaño $n - 1$ cuando el tamaño del problema general es n. Deduzca la *desigualdad*

$$T(n) \geq 2T(n - 1) + n$$

y determine una cota inferior para su solución.

10.5 Suponga que las dimensiones de las matrices A, B, C y D son 20×2, 2×15, 15×40 y 40×4, respectivamente. Se quiere hallar la mejor forma de calcular $A \times B \times C \times D$. Muestre los arreglos `costo`, `ultimo` y `ordenMult` calculados por los algoritmos 10.1 y 10.2.

10.6 Sean $A_1, \ldots, A_n$ matrices tales que las dimensiones de A_i son $d_{i-1} \times d_i$, para $i = 1, \ldots, n$. He aquí una propuesta para un algoritmo codicioso que determina el mejor orden en que se puede efectuar la multiplicación de matrices $A_1 \times A_2 \times \cdots \times A_n$.

```
ordenCodicioso(dim, n)   // BOSQUEJO
    En cada paso, escoger la dimensión restante más grande (de entre dim[1], ..., dim[n-1]
    y multiplicar dos matrices adyacentes que compartan esa dimensión.
```

Observe que la estrategia produce el orden óptimo de multiplicación para las matrices del ejemplo 10.4.

a. ¿Qué orden tiene el tiempo de ejecución de este algoritmo (sólo para determinar el orden en que se multiplicarán las matrices, sin incluir las multiplicaciones mismas)?
b. Presente un argumento convincente para afirmar que esta estrategia siempre obtiene el número mínimo de multiplicaciones, o bien dé un ejemplo en el que no lo haga.

10.7 Elabore un ejemplo con sólo tres o cuatro matrices en el que el peor orden de multiplicación efectúe por lo menos 100 veces más multiplicaciones de elementos que el mejor orden.

Sección 10.4 Construcción de árboles de búsqueda binaria óptimos

10.8 Suponga que en un árbol de búsqueda binaria totalmente equilibrado con $n = 2^k - 1$ nodos todas las claves tienen la misma probabilidad de ser buscadas. Deduzca una expresión para el número medio de comparaciones que se requieren para hallar una clave, suponiendo que se cuenta con comparaciones de tres vías. (Cabe señalar que el peor caso es $k = \lg(n + 1)$.)

10.9

a. Calcule los valores contenidos en las matrices `costo` y `raiz` del algoritmo de programación dinámica para determinar el árbol de búsqueda binaria óptimo (algoritmo 10.3) para las claves siguientes. (La probabilidad de cada clave se da entre paréntesis.)

$$A(0.20), \quad B\ (0.24), \quad C(0.16), \quad D(0.28), \quad E(0.04), \quad F(0.08)$$

b. Dibuje el árbol óptimo.

10.10 Suponga que se ejecutó el algoritmo 10.3 con las claves $K_1, \ldots, K_n$ cuyas probabilidades son $p_1, \ldots, p_n$. Escriba un algoritmo que utilice el arreglo `raiz` calculado por el algoritmo 10.3 para construir el árbol de búsqueda binaria óptimo. Utilice el TDA `ArbolBin` de la sección 2.3.3 para construir el resultado. ¿Qué orden asintótico tiene el tiempo de ejecución de su algoritmo? (Deberá estar en $O(n)$.) *Sugerencia:* Estudie el algoritmo 10.2.

★ **10.11** Muestre cómo puede calcularse $p(i, j)$, empleado en el algoritmo 10.3, en $\Theta(1)$ por invocación, después de un preprocesamiento $\Theta(n)$.

10.12 Describa un algoritmo codicioso directo para el problema de construir árboles de búsqueda binaria óptimos. ¿Siempre produce el árbol óptimo? Justifique su respuesta con un argumento o un contraejemplo.

Sección 10.5 División de sucesiones de palabras en líneas

10.13 Muestre que el algoritmo codicioso para dividir líneas que se mencionó en la sección 10.5 no produce el castigo mínimo en todos los casos.

10.14

a. Determine una división en líneas óptima para el ejemplo 10.8 utilizando $\mathcal{DP}$(`divLineas`).

b. ¿Cuántos subproblemas es preciso evaluar?

★ **c.** ¿Cuántos subproblemas se evaluarían empleando `divLineas` en su forma recursiva natural? *Sugerencia:* Utilice programación dinámica para contar las invocaciones que efectúa la forma recursiva natural.

10.15 Complete el bosquejo del algoritmo para dividir líneas de la sección 10.5. La salida del algoritmo general deberá ser un arreglo `ultimaPalabra` tal que (una vez que el algoritmo termi-

na) `ultimaPalabra[L]` es el índice de la última palabra que se coloca en la línea *L*. (Si `ultimaPalabra[L]` es *n*, el párrafo termina en la línea *L*.)

Problemas adicionales

10.16 Los coeficientes binomiales se pueden definir con la ecuación de recurrencia

$$\begin{aligned} C(n, k) &= C(n-1, k-1) + C(n-1, k) && \text{para } n > 0 \text{ y } k > 0 \\ C(n, 0) &= 1 && \text{para } n \geq 0 \\ C(0, K) &= 0 && \text{para } k > 0. \end{aligned}$$

$C(n, k)$ también se pronuncia "*n* escoger *k*" y se denota con $\binom{n}{k}$. Se trata del número de formas en que podemos escoger *k* objetos distintos de un conjunto de *n* objetos. (Vea la ecuación (1.1) y el ejercicio 1.2.) Considere las cuatro formas que siguen de calcular $C(n, k)$ para $n \geq k$.

1. Una función recursiva sugerida por la relación de recurrencia dada para $C(n, k)$.
2. Un algoritmo de programación dinámica.
3. La fórmula $C(n, k) = \dfrac{n(n-1)\cdots(n-k+1)}{k!}$.
4. La fórmula $C(n, k) = \dfrac{n!}{k!(n-k)!}$.

Evalúe estos métodos como sigue.

a. Escriba un bosquejo de cada método para cerciorarse de entender qué trabajo debe efectuarse con cada uno.

b. Compare la cantidad de trabajo realizada por cada método. Indique qué operaciones está contando. Compare la cantidad de espacio ocupada por cada método.

c. ¿Cualquiera de los cuatro métodos tiene otras ventajas o desventajas importantes? (Por ejemplo, ¿alguno de ellos tiene más probabilidades de causar un error por desbordamiento aritmético?, ¿y qué hay con el truncado causado por la división entera?)

10.17 Sea *E* un arreglo de *n* enteros distintos. Escriba un algoritmo para determinar la longitud de una subsucesión creciente de elementos de *E* que tenga longitud máxima. No es preciso que la subsucesión sea contigua en la sucesión original. Por ejemplo, si los elementos son 11, 17, 5, 8, 6, 4, 7, 12, 3, una subsucesión creciente de longitud máxima es 5, 6, 7, 12. Analice el tiempo de ejecución de peor caso y las necesidades de espacio de su algoritmo.

10.18 Dos cadenas de caracteres podrían tener muchas subcadenas en común. Es obligatorio que las subcadenas sean contiguas en la cadena original. Por ejemplo, *fotografía* y *tomografía* tienen varias subcadenas de longitud 1 (es decir, letras individuales) en común, y las subcadenas comunes *to* y *ografía* (además de todas las subcadenas de *ografía*). La longitud de subcadena común máxima es 7.

Sean $X = x_1x_2\cdots x_n$ y $Y = y_1y_2\cdots y_m$ dos cadenas de caracteres. Escriba un algoritmo que determine la longitud de subcadena común máxima para *X* y *Y*. Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo en función de *n* y de *m*. *Nota:* Existe una so-

lución de programación dinámica $\Theta(nm)$, así como otras soluciones $\Theta(nm)$ que no usan programación dinámica. Trate de hallar dos soluciones.

10.19 Sean A y B arreglos de n enteros cada uno. Una *subsucesión común* de A y B es una sucesión que es una subsucesión de A y es una subsucesión de B. La subsucesión no tiene que ser contigua en A ni en B. Por ejemplo, si los elementos de A son 5, 8, 6, 4, 7, 1, 3 y los elementos de B son 4, 5, 6, 9, 7, 3, 2, una subsucesión común de longitud máxima es 5, 6, 7, 3. Escriba un algoritmo para hallar una subsucesión común de longitud máxima de A y B. Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo.

10.20 Suponga que le dan tres cadenas de caracteres: $X = x_1x_2 \cdots x_m$, $Y = y_1y_2 \cdots y_n$ y $Z = z_1z_2 \cdots z_{m+n}$. Decimos que Z es un *barajado* de X y Y si podemos formar Z intercalando los caracteres de X y Y de modo que se mantenga el ordenamiento de izquierda a derecha de los caracteres de cada cadena. Por ejemplo, *catliabernaciastas* es un barajado de *calabacitas* y *tiernas*, pero *calatinerbacasitas* no lo es. Idee un algoritmo de programación dinámica que reciba como entradas X, Y, Z, m y n, y determine si Z es o no un barajado de X y Y. Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo. *Sugerencia:* Los valores de su diccionario deben ser booleanos, no numéricos.

★ **10.21** El *problema de partición* consiste en, dada una sucesión de n números no negativos como entrada, hallar una forma de dividir esta sucesión en dos subsucesiones disjuntas de modo que las sumas de los enteros de cada una de las subsucesiones sean iguales. En términos más formales, dados los enteros no negativos $s_1, \ldots, s_n$ cuya suma es S, hallar un subconjunto I de $\{1, 2, \ldots, n\}$ tal que

$$\sum_{i \in I} s_i = \sum_{j \notin I} s_j = S/2,$$

o determinar que no existe tal subconjunto. Escriba un algoritmo de programación dinámica para el problema de partición. Analice el tiempo de ejecución y las necesidades de almacenamiento de peor caso de su algoritmo en función de n y S.

★ **10.22** Suponga que tiene n dólares que invertir en cualquiera de m empresas. Suponga que n es entero y que todas las inversiones deben ser cantidades enteras. La tabla de entrada `rendimInv` describe los rendimientos esperados de inversiones individuales. Específicamente, `rendimInv[d][j]` es el rendimiento esperado de d dólares en la empresa j.

a. Escriba un algoritmo para determinar el rendimiento máximo que puede esperar si invierte n dólares. (Puede suponer que las columnas de `rendimInv` son no decrecientes; es decir, que invertir más dinero en una empresa no hará que baje el rendimiento que se obtiene de esa empresa.)

b. Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo en función de n y m.

c. Amplíe su algoritmo para determinar el plan de inversión óptimo. (Haga lo que sea necesario para averiguar cuánto conviene invertir en cada empresa.) Analice el tiempo de ejecución y las necesidades de espacio de peor caso.

d. Suponga que no puede hacer el supuesto de la parte (a). Dicho de otro modo, suponga que invertir dinero adicional en una empresa puede reducir el rendimiento total que se obtiene de esa empresa. Demuestre que su algoritmo ya funciona correctamente en tales casos o bien dé un ejemplo en el que no calcule el rendimiento máximo posible. Luego indique cómo modificarlo para que funcione correctamente en general.

10.23 Suponga que las denominaciones de las monedas de un país son $c_1 > c_2 > \cdots > c_n$ (por ejemplo, 50, 25, 10, 5, 1 para Estados Unidos). El *problema de cambio en monedas* consiste en, dada una sucesión de denominaciones de monedas y un importe a en centavos, determinar el número mínimo de monedas que se necesitan para dar a centavos de cambio. (Puede suponer que $c_n = 1$, así que siempre es posible dar cambio para cualquier cantidad a.)

a. Describa un algoritmo codicioso para este problema. Explique cómo operaría para dar $1.43 (dólares) de cambio.

b. Demuestre que su algoritmo codicioso funciona con monedas estadounidenses; es decir, que da cambio con el menor número posible de monedas.

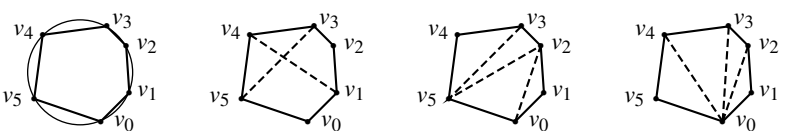
c. Invente un ejemplo de denominaciones para el sistema monetario de un país ficticio con el que su algoritmo codicioso no dé el número mínimo de monedas.

d. Escriba un algoritmo de programación dinámica para resolver el problema. Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo en función de n y a.

10.24 Escriba un algoritmo para determinar cuántas formas distintas hay de dar a centavos de cambio utilizando monedas de 50, 25, 10, 5 y 1 centavos. Por ejemplo, hay seis formas de dar 17 centavos de cambio: $10 + 5 + 1 + 1$; $10 + (7 \times 1)$; $5 + 5 + 5 + 1 + 1$; $5 + 5 + (7 \times 1)$; $5 + (12 \times 1)$ y 17 monedas de centavo.

★ **10.25** Un *polígono* de n lados es un grafo no dirigido con n vértices y n aristas que forman un ciclo simple, $v_0, v_1, \ldots, v_{n-1}, v_0$. (Por convención, los vértices de un polígono se indizan a partir de 0.) Una *cuerda* de un polígono es una arista (no dirigida) entre cualesquier dos vértices no adyacentes del polígono. Dos cuerdas distintas, digamos wx y yz, son *no intersecantes* si existe un camino de aristas de polígono de w a x que no contenga y ni z como vértice intermedio. Si dos cuerdas comparten exactamente un vértice, son no intersecantes. Una *triangulación* de un polígono es un conjunto máximo de cuerdas mutuamente no intersecantes. Un *polígono triangulado* es el grafo que consiste en el polígono original y un conjunto de cuerdas que constituyen una triangulación de ese polígono.

Dimos estas definiciones pensando en que los vértices del polígono están ubicados en una sucesión antihoraria (en el sentido contrario al giro de las manecillas del reloj) alrededor de un círculo, pero las definiciones no requieren tal ubicación. La figura que sigue muestra un polígono con sus vértices en un círculo, un ejemplo de cuerdas intersecantes y dos posibles triangulaciones.



Suponga que todas las posibles cuerdas tienen asociados pesos reales. Por ejemplo, si los vértices del polígono son puntos en el espacio, el peso de una cuerda podría ser la distancia entre sus dos vértices, pero puede haber otros esquemas de ponderación. El problema para este ejercicio es, dado un polígono y un conjunto de pesos para sus cuerdas como entrada, hallar una *triangulación de peso mínimo*, es decir, una que minimice la suma de los pesos de las cuerdas.

a. Demuestre que cada arista del polígono está en exactamente un triángulo de un polígono triangulado.

★ **b.** Diseñe un algoritmo de programación dinámica que resuelva el problema de la triangulación de peso mínimo de un polígono en general. Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo en función de n.

★ **c.** Suponga ahora que los vértices del polígono están en posiciones dadas sobre la circunferencia de un círculo y que el peso de una cuerda $v_i v_j$ es la magnitud del arco circular entre v_i y v_j, en grados. Por ejemplo, si una cuerda separa una cuarta parte del círculo, su peso será 90. Así, todos los pesos son positivos y no pueden ser mayores que 180. ¿Puede diseñar un algoritmo para este caso especial que se ejecute más rápidamente que el de la parte (b)? Analice el tiempo de ejecución y las necesidades de espacio de peor caso de su algoritmo en función de n.

★ **10.26** Suponga que ha heredado los derechos a 500 canciones no estrenadas, grabadas por el popular grupo Roqueros Roncos. Usted planea sacar un conjunto de cinco discos compactos (numerados del 1 al 5) con algunas de esas canciones. Cada disco puede contener cuando más 60 minutos de música y ninguna canción puede continuar de un disco al siguiente. Puesto que usted es aficionado a la música clásica y no tiene cómo juzgar el mérito artístico de estas canciones, decide usar los criterios siguientes para seleccionar las piezas:

1. Las canciones se grabarán en el juego de discos en orden según la fecha en que se escribieron.
2. Se incluirá el número máximo posible de canciones.

Suponga que tiene una lista de las longitudes de las canciones, $l_1, l_2, \ldots, l_{500}$, en orden según la fecha en que se escribieron. (Ninguna canción dura más de 60 minutos.)

Escriba un algoritmo para determinar el número máximo de canciones que se pueden incluir en el juego de discos satisfaciendo los criterios dados. *Sugerencia:* Sea `T[i][j]` el tiempo mínimo que requieren cualesquier i canciones seleccionadas de entre las primeras j canciones. Se deberá interpretar T de modo que incluya el tiempo en blanco, si lo hay, al final de un disco terminado. Dicho de otro modo, si una selección de canciones ocupa un disco más los primeros 15 minutos de un segundo disco, el tiempo de esa selección se toma como 75 minutos, aunque haya unos cuantos minutos en blanco al final del primer disco.

Programas

1. Escriba un programa que construya un árbol de búsqueda binaria óptimo utilizando el algoritmo 10.3 y su solución al ejercicio 10.10.
2. Escriba un programa para efectuar la división en líneas de las palabras de un párrafo. Implemente varias estrategias, incluida la estrategia codiciosa simple y la solución de programación dinámica con castigo mínimo. Pruebe algunas variantes para la función `costoLinea`, como

la que usamos en el texto, X^3, y $(X/k)^3$, donde X es el número de espacios extra y k es el número de palabras en la línea. Para el castigo total, pruebe determinar el mínimo de `costoLinea` de entre todas las líneas del párrafo, en lugar de la suma.

Notas y referencias

Bellman (1957) y Bellman y Dreyfus (1962) son referencias de programación dinámica estándar desde el punto de vista de la teoría de control.

Si se desea un tratamiento mucho más extenso de los árboles de búsqueda binaria óptimos, véase Knuth (1998). Yao (1982) describe técnicas para acelerar las soluciones de programación dinámica y contiene algoritmo $O(n^2)$ para los problemas del orden de multiplicación de matrices y árbol de búsqueda binaria que cubrimos en las secciones 10.3 y 10.4.

El ejercicio 10.26 es una contribución de J. Frankle.

Thompson (1986) describe el uso de la programación dinámica para resolver finales de partidas de ajedrez con un conjunto específico de piezas en el tablero trabajando hacia atrás desde todas las posiciones de *jaque mate* posibles. Bentley (1986) también resume el trabajo. En Thompson (1990, 1991 y 1996) aparecen resultados posteriores.

11

Cotejo de cadenas

11.1 Introducción

En este capítulo estudiaremos el problema de detectar la ocurrencia de una subcadena específica, llamada *patrón*, en otra cadena, llamada *texto*. El problema suele presentarse en el contexto de las cadenas de caracteres y surge a menudo en el procesamiento de textos, así que supondremos este contexto en nuestras explicaciones y ejemplos. No obstante, las soluciones aquí presentadas se pueden aplicar a otros contextos, como el cotejo de una serie de bytes que representan datos gráficos, código de máquina u otros datos, y el cotejo de una sublista de una lista ligada. Los primeros tres algoritmos que describiremos en este capítulo buscan una coincidencia exacta con el patrón. El problema del cotejo aproximado se tratará en la sección 11.5. Usaremos la nomenclatura siguiente en todo el capítulo.

Definición 11.1 Notación para patrones y texto

Este capítulo utiliza las convenciones de notación siguientes.

P	El patrón que se busca
T	El texto en el que se busca P
m	La longitud de P
n	La longitud de T, que el algoritmo desconoce; sólo se usa para análisis
p_i, t_i	Los i-ésimos caracteres de P y T se denotan con letras minúsculas y subíndices. Se supone que el primer índice tanto de P como de T es 1
j	Posición actual dentro de T
k	Posición actual dentro de P

Suponemos que se nos da una función **`boolean`** que nos dice si hemos rebasado el último carácter del texto: `finTexto`(T, j) devuelve **`true`** si j es mayor que el índice del último carácter de T y **`false`** en caso contrario. ■

En el seudocódigo de este capítulo suponemos que tanto P como T son arreglos de caracteres. Este supuesto es razonable en el caso de P porque lo normal es que sea relativamente corto y esté disponible para que los algoritmos de cotejo de cadenas lo sometan a un procesamiento previo. En cambio, T bien podría ser de un tipo distinto, podría ser en extremo largo y podría no estar todo disponible en la memoria. No obstante, veremos que los algoritmos sólo efectúan unas cuantas operaciones con T, y no aprovechan toda la flexibilidad del acceso a los arreglos, así que pueden adaptarse fácilmente a aplicaciones en las que T no es un arreglo. Abordaremos algunas de estas cuestiones en los ejercicios. Supondremos que n es relativamente grande en comparación con m. Como se usa `finTexto`, los algoritmos no necesitan conocer n, pero sí se le usa en los análisis.

Detalle de Java: Java proporciona una clase integrada llamada **`String`** (cadena), que no es lo mismo que un arreglo de caracteres. En aras de la independencia respecto al lenguaje, no usaremos esta clase integrada.

Recomendamos al lector meditar el problema del cotejo de cadenas y escribir, o al menos bosquejar, un algoritmo para resolverlo antes de continuar. Su algoritmo seguramente será muy parecido al primero que presentaremos, que es relativamente directo.

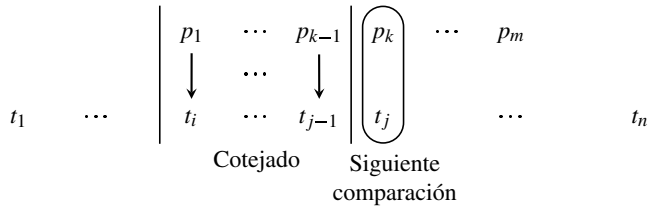


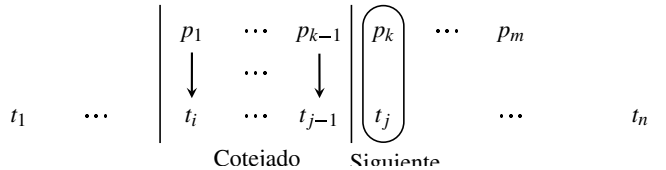
Figura 11.1 Panorama general del algoritmo 11.1

11.2 Una solución directa

Examinemos primero un procedimiento muy sencillo para cotejar cadenas. Comenzando por el principio de cada cadena, comparamos caracteres uno tras otro hasta que el patrón se agota o bien se halla una diferencia. En el primer caso ya terminamos; se ha hallado una copia del patrón en el texto. En el segundo caso volvemos a comenzar, comparando el primer carácter del patrón con el segundo carácter del texto. En general, cuando se halla una diferencia, deslizamos (de manera figurada) el patrón una posición más hacia adelante sobre el texto y volvemos a comenzar, comparando el primer carácter del patrón con el siguiente carácter del texto.

Ejemplo 11.1 Cotejo directo de cadenas

Se efectúan comparaciones (de izquierda a derecha) entre los pares de caracteres indicados por flechas. Si hay alguna diferencia, el patrón se desplaza una posición hacia adelante respecto al texto, y las comparaciones se reinician en el extremo izquierdo del patrón.



Obsérvese que si el patrón se desplaza hasta después del punto en el que se detectó la primera diferencia, podría no detectarse una ocurrencia del patrón. ■

Algoritmo 11.1 Cotejo directo de cadenas

Entradas: P y T, las cadenas del patrón y del texto; m, la longitud de P. Se supone que el patrón no está vacío.

Salidas: El valor devuelto es el índice de T donde principia una copia de P, o -1 si no se encuentra el patrón en el texto.

Comentarios: El panorama general se muestra en la figura 11.1. El algoritmo no necesita realmente la variable de índice i porque se puede calcular a partir de j y k (es decir, $i = j - k + 1$). La función `finTexto` es la que se definió en la definición 11.1.

```
int cotejoSimple(char[] P, char[] T, int m)
    int coincide;  // valor a devolver
    int i, j, k;
    // i es donde se conjetura actualmente que P principia en T;
    // j es el índice del carácter actual en T;
    // k es el índice del carácter actual en P.
    coincide = −1;
    j = 1; k = 1;
    i = j;
    while (finTexto(T, j) == false)
        if (k > m)
            coincide = 1;  // Se halló el patrón
            break;

        if (T[j] == P[k])
            j ++; k ++;
        else
            // Retroceder sobre los caracteres que coincidieron.
            int retroceso = k − 1;
            j = j − retroceso; k = k − retroceso;
            // Deslizar el patrón hacia adelante, volver a comenzar.
            j ++;
            i = j;
        // Continuar el ciclo.
    return coincide;
```

Análisis

Contaremos las comparaciones de caracteres efectuadas por nuestros algoritmos de cotejo de cadenas. Esto es ciertamente razonable en el caso del algoritmo 11.1, dada su sencilla estructura cíclica. Hay unos cuantos casos fáciles. Si el patrón aparece al principio del texto, se efectúan m comparaciones. Si p_1 no está en T, se efectúan n comparaciones. ¿Cuál es el peor caso? El número de comparaciones sería máximo si para cada valor de i —es decir, para cada posible posición de inicio de P en T— todos los caracteres de P excepto el último coincidieran con los caracteres correspondientes del texto. Entonces, el número de comparaciones de caracteres en el peor caso es cuando más mn, y la complejidad del algoritmo está en $O(mn)$.

Con algunos algoritmos, entradas que requieren mucho trabajo en un paso podrían requerir muy poco trabajo en otro paso. Por tanto, si acumulamos el máximo trabajo posible en cada paso obtendremos una cota superior pero no necesariamente un valor exacto para el trabajo que se efectúa en el peor caso.

Para demostrar que el peor caso *requiere* (aproximadamente) mn comparaciones (es decir, para demostrar que la complejidad de peor caso está en $\Theta(mn)$), deberemos demostrar que la situación descrita sí puede presentarse, es decir que es posible construir P y T de modo que todos los caracteres de P excepto el último coincidan con los caracteres correspondientes iniciando en cualquier punto de T. Sea P = '$AA \cdots AB$' ($m - 1$ Aes seguidas de una B) y T = '$A \cdots A$' (n Aes).

Este ejemplo de peor caso no se presenta con frecuencia en los textos en lenguaje natural. De hecho, el algoritmo 11.1 funciona muy bien en promedio con los lenguajes naturales. En algunos estudios empíricos el algoritmo efectuó sólo aproximadamente 1.1 comparaciones de caracteres por cada carácter de *T* (hasta el punto en que se halló el patrón o hasta el final de *T* si no se halló). Ello implica que pocos caracteres del texto tuvieron que examinarse más de una vez.

El algoritmo 11.1 tiene una propiedad que en algunas aplicaciones es indeseable: podría ser necesario retroceder a menudo en la cadena de texto (una distancia `retroceso` − 1, pues se resta `retroceso` a *j* y se le suma 1 para iniciar una nueva búsqueda del patrón en el ciclo **`while`**). Si el texto se está leyendo de una fuente de entrada que no permite retroceder, ello dificultaría el uso del algoritmo (en el ejercicio 11.4 se sugiere una solución que usa el espacio de manera eficiente). El algoritmo que presentamos en la sección siguiente se ideó específicamente para no tener que retroceder en el texto y resultó además ser más rápido (en el peor caso).

11.3 El algoritmo Knuth-Morris-Pratt

Primero describiremos brevemente, sin algoritmos formales, un enfoque del problema de cotejo de patrones que tiene algunas cualidades importantes pero también algunas desventajas. La construcción empleada por el algoritmo principal de esta sección fue sugerida por el método que describiremos a continuación y rescata algunas de sus ventajas al tiempo que elimina las desventajas.

11.3.1 Cotejo de patrones con autómatas finitos

Dado un patrón *P*, es posible construir un *autómata finito* que sirva para buscar con gran rapidez una copia de *P* en el texto. Podemos interpretar fácilmente un autómata finito como un tipo especial de máquina o de diagrama de flujo, pero no es necesario tener conocimientos de teoría de autómatas para entender este método.

Definición 11.2

Sea Σ el alfabeto, o conjunto de caracteres, del cual pueden escogerse los caracteres de *P* y de *T*, y sea $\alpha = |\Sigma|$. El diagrama de flujo, o autómata finito, tiene dos tipos de nodos:

1. Algunos nodos *de lectura* que significan: "Leer el siguiente carácter del texto. Si no hay más caracteres en la cadena de texto, parar; no hay coincidencia". Un nodo *de lectura* se designa como nodo *de inicio*.
2. Un nodo *de paro* que significa: "Parar; se halló una coincidencia". Se le marca con un asterisco. ■

El diagrama de flujo tiene α flechas que salen de cada nodo *de lectura*. Cada flecha está rotulada con un carácter de Σ. La flecha que coincide con el carácter de texto que se acaba de leer es la flecha a seguir; es decir, indica el nodo que se visitará a continuación. Recomendamos al lector estudiar el ejemplo de la figura 11.2 para entender por qué las flechas apuntan a donde lo hacen. Los nodos *de lectura* sirven como una especie de memoria. Por ejemplo, si la ejecución llega al tercer nodo *de lectura*, es porque los últimos dos caracteres leídos del texto fueron *A*. Lo que vino antes no tiene importancia. Para que haya una coincidencia, deben seguir inmediatamente una *B* y una *C*. Si el siguiente carácter es una *B*, podremos pasar al nodo 4, que recuerda que

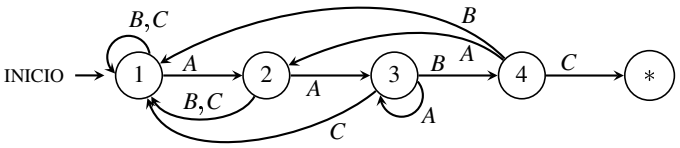


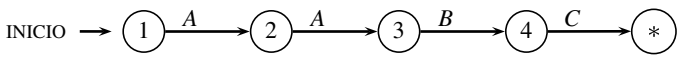
Figura 11.2 Autómata finito para $P =$ '*AABC*'

ya apareció *AAB*. Por otra parte, si el siguiente carácter leído en el nodo 3 hubiera sido una *C*, tendríamos que haber regresado al nodo 1 y esperar a que aparezca otra *A* para iniciar el patrón.

Una vez construido el diagrama de flujo para el patrón, se puede probar el texto para ver si ocurre en él el patrón examinando cada carácter del texto una sola vez y por ende en tiempo $O(n)$. Esto es mucho mejor que el algoritmo 11.1, tanto en términos de tiempo como por el hecho de que una vez examinado un carácter del texto, nunca tendrá que volverse a considerar; no hay retroceso en el texto. La dificultad radica en la construcción del autómata finito; es decir, en decidir a dónde apuntan todas las flechas. Existen algoritmos muy conocidos para construir un autómata finito que reconozca un patrón dado, pero en el peor caso esos algoritmos requieren mucho tiempo. La dificultad se debe al hecho de que de cada nodo *de lectura* sale una flecha para cada carácter de Σ. Se requiere tiempo para determinar a dónde debe apuntar cada flecha y espacio para representar $m\alpha$ flechas. Por tanto, un mejor algoritmo tendría que eliminar algunas de las flechas.

11.3.2 El diagrama de flujo Knuth-Morris-Pratt

Al construir el autómata finito para un patrón *P*, es fácil incluir las flechas que corresponden a una coincidencia. Por ejemplo, al dibujar la figura 11.2 para el patrón '*AABC*', el primer paso es dibujar,



la parte difícil es la inserción del resto de las flechas. El algoritmo Knuth-Morris-Pratt (que, para abreviar, llamaremos algoritmo KMP) también construye una especie de diagrama de flujo que se usará para examinar el texto. El diagrama de flujo KMP contiene las flechas fáciles —es decir, las que deben seguirse si se lee del texto el carácter deseado— pero sólo contiene una flecha más que sale de cada nodo: la flecha a seguir si el carácter deseado *no* se leyó del texto. Llamamos a las flechas ligas de éxito y ligas de fracaso, respectivamente. El diagrama de flujo KMP difiere del autómata finito en varios detalles: los rótulos de caracteres del diagrama de flujo KMP están en los nodos, no en las flechas; se lee el siguiente carácter del texto sólo después de haber seguido una liga de éxito; si se sigue una liga de fracaso, se volverá a considerar el mismo carácter del texto. Hay un nodo extra que hace que se lea un nuevo carácter del texto; la exploración se inicia en este nodo. Al igual que en el autómata finito, si se llega al nodo rotulado con *, se habrá hallado una copia del patrón; si se llega al final del texto en cualquier otro punto del diagrama de flujo, la exploración terminará sin éxito. Esta descripción formal del procedimiento de exploración deberá permitir al lector usar el diagrama de flujo KMP de la figura 11.3

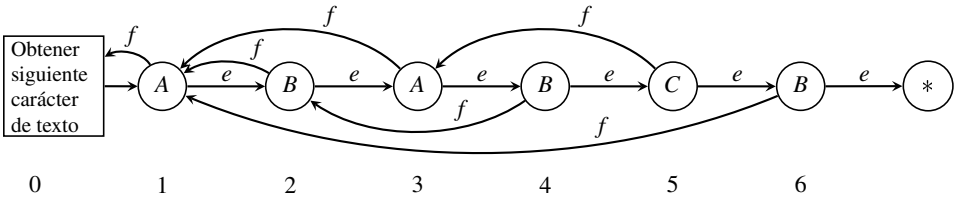


Figura 11.3 Diagrama de flujo KMP para P = '*ABABCB*'

Número de celda KMP	*Texto explorado*		Éxito (e) o fracaso (f)
	Índice	Carácter	
1	1	*A*	e
2	2	*C*	f
1	2	*C*	f
0	2	*C*	obt. sig. car.
1	3	*A*	e
2	4	*B*	e
3	5	*A*	e
4	6	*A*	f
2	6	*A*	e
1	6	*A*	s
2	7	*B*	e
3	8	*A*	e
4	9	*B*	e
5	10	*A*	f
3	10	*A*	e
4	11	ninguno	fracaso

Tabla 11.1 Acción del diagrama de flujo KMP de la figura 11.3 para el patrón '*ABABCB*' en el texto '*ACABAABABA*'

para explorar una cadena de texto. Pruebe '*ACABAABABA*' y consulte la tabla 11.1 si se topa con problemas.

Ahora necesitamos una representación en computadora del diagrama de flujo KMP, un algoritmo para construirlo (para determinar la forma de colocar las ligas de fracaso), un algoritmo formal para el procedimiento de exploración y un análisis de los dos algoritmos.

11.3.3 Construcción del diagrama de flujo KMP

La representación del diagrama de flujo es muy sencilla; usamos dos arreglos, uno para contener los caracteres del patrón y uno para contener las ligas de fracaso. Las ligas de éxito están implícitas en el ordenamiento de los elementos del arreglo.

Sea `fracaso` el arreglo de ligas de fracaso; `fracaso[k]` será el índice del nodo al que apunta la liga de fracaso del k-ésimo nodo, para $1 \leq k \leq m$. El nodo especial que simplemente hace que se lea el siguiente carácter se considera el nodo cero; `fracaso[1]` $= 0$. Para ver cómo establecer las otras ligas de fracaso, consideraremos un ejemplo.

Ejemplo 11.2 Establecimiento de ligas de fracaso para el algoritmo KMP

Sea $P =$ '*ABABABCB*' y supóngase que los primeros seis caracteres han coincidido con seis caracteres consecutivos del texto, como se indica:

```
P :        |A B A B A B|C B
           |↓ ↓ ↓ ↓ ↓ ↓|
T : . . .  |A B A B A B|x  . . .
```

Supóngase que el siguiente carácter del texto, x, no es una '*C*'. El siguiente lugar del texto en el que el patrón podría principiar es en la tercera posición que se muestra, es decir:

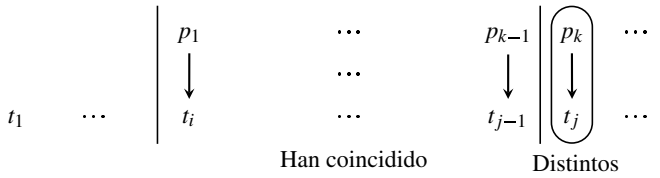
```
P :              |A B A B|A B C B

T : . . .  A B  |A B A B|x  . . .
```

El patrón se desplaza hacia adelante de modo que el segmento inicial más largo que coincida con parte del texto que precede a x quede alineado con esa parte del texto. Ahora se deberá probar x para ver si es una A que coincida con la tercera A del patrón. Por tanto, la liga de fracaso del nodo que contiene la C deberá apuntar al nodo que contiene la tercera A. ■

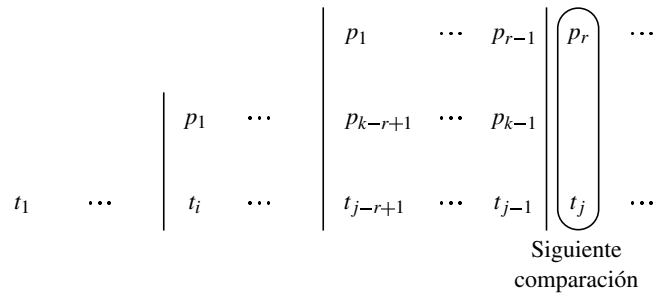
El panorama general se muestra en la figura 11.4. Cuando se encuentra una diferencia, hay que deslizar P hacia adelante, pero manteniendo el traslapo más largo posible de un prefijo de P con un sufijo de la parte del texto que ha coincidido con el patrón hasta ese momento. Por tanto, el carácter actual del texto se deberá comparar a continuación con p_r; es decir, `fracaso[k]` deberá ser r. Sin embargo, queremos construir el diagrama de flujo antes de ver T. ¿Cómo determinamos r sin conocer T? La observación clave es que, cuando exploremos T, la parte de T que acabamos de explorar habrá coincidido con la parte de P que acabamos de explorar, así que sólo tendremos que hallar el traslapo más grande de un prefijo de P con un sufijo de la parte de P que acabamos de explorar.

Definición 11.3 Ligas de fracaso

Definimos `fracaso[k]` como el r más grande (con $r < k$) tal que $p_1 \cdots p_{r-1}$ coincide con $p_{k-r+1} \cdots p_{k-1}$. Es decir, el *prefijo* de $r - 1$ caracteres de P es idéntico a la subcadena de $r - 1$ caracteres que *termina* en el índice $k - 1$. Por tanto, las ligas de fracaso se determinan mediante repetición dentro del mismo P. ■



(a) Diferencia en p_k y t_j



(b) Deslizamos p hasta alinear el prefijo más largo
que coincida con un sufijo de los caracteres explorados

Figura 11.4 Deslizamiento del patrón para el algoritmo KMP

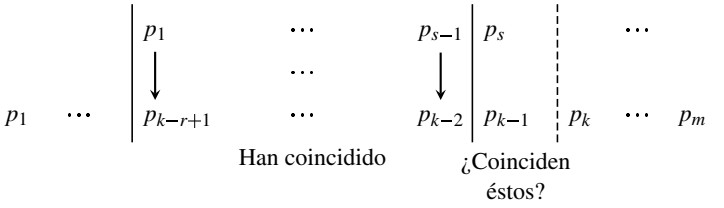
Podríamos no detectar una ocurrencia del patrón si escogemos un r demasiado pequeño. (Consideremos lo que sucedería si en el ejemplo 11.2 se estableciera la liga de fracaso de C de modo que apuntara a la segunda A, y si $x = A$ seguida de BCB en el texto.)

Aunque hemos descrito los valores correctos para las ligas de fracaso, todavía no tenemos un algoritmo eficiente para calcularlas. Podemos definir `fracaso` de forma recursiva. Supóngase que ya hemos calculado las primeras $k-1$ ligas de fracaso. Entonces tendremos la situación de la figura 11.5(a). Para asignar `fracaso[k]` necesitaremos cotejar una subcadena de P que termina en $k-1$. A fin de simplificar la notación, usaremos $s =$ `fracaso[k-1]`. El caso fácil es cuando $p_{k-1} = p_s$, porque ya sabemos que $p_1 \cdots p_{s-1}$ coincide con la subcadena de $s-1$ caracteres que termina en $k-2$. Entonces podremos extender en un carácter más las dos sucesiones coincidentes de la figura 11.5(a), así que en este caso asignamos $s+1$ a `fracaso[k]`.

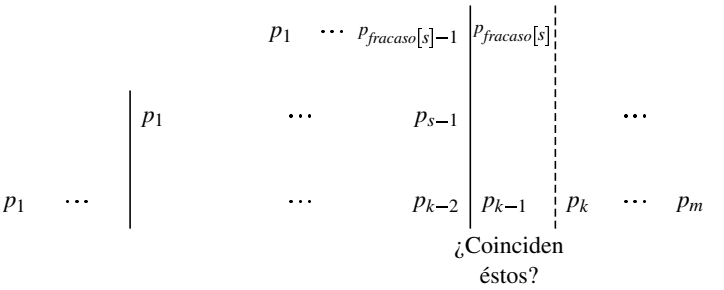
Ejemplo 11.3 Cálculo de ligas de fracaso KMP: un caso sencillo

En la figura 11.6, `fracaso[6]` $= 4$ porque $p_1p_2p_3$ coincide con $p_3p_4p_5$. Puesto que $p_6 = p_4$, asignamos 5 a `fracaso[7]`. Esto nos dice que $p_1 \cdots p_4$ coincide con la subcadena de cuatro caracteres que termina en el índice 6. ■

¿Qué sucede si $p_{k-1} \neq p_s$? Deberemos hallar un prefijo de P que coincida con una subcadena que termine en $k-1$. En este caso no puede extenderse la coincidencia de la figura 11.5(a), así que buscamos más atrás. Sea $s_2 =$ `fracaso[s]`. Por las propiedades de las ligas de fracaso



(a) Por definición de `fracaso[k-1]` (que es s)



(b) Buscamos hacia atrás una coincidencia con p_{k-1}

Figura 11.5 Cálculo de ligas de fracaso: el índice s es igual a `fracaso[k−1]`

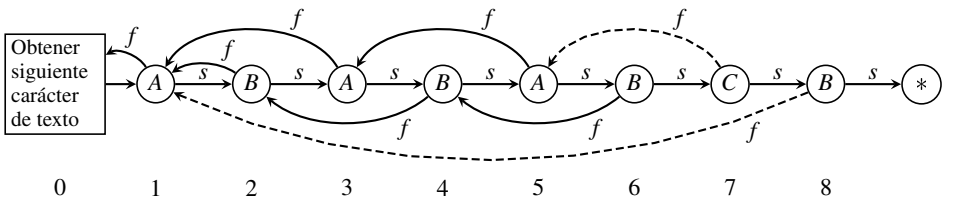


Figura 11.6 Cálculo de ligas de fracaso: las aristas punteadas se explican en los ejemplos 11.3 y 11.4

tenemos las coincidencias que se muestran en la figura 11.5(b). Si $p_{k-1} = ps_2$, tendremos un prefijo (de longitud s_2) que coincide con la subcadena que termina en $k - 1$, así que `fracaso[k]` deberá ser $s_2 + 1$. Si $p_{k-1} \neq ps_2$, deberemos seguir la liga de fracaso que sale del nodo s_2 y volver a intentarlo. Este proceso continúa hasta que hallamos una liga de fracaso s tal que $s = 0$ o $p_{k-1} = p_s$. En ambos casos, `fracaso[k]` deberá ser $s + 1$.

Ejemplo 11.4 Cálculo de ligas de fracaso KMP: el caso recursivo

Examinemos otra vez la figura 11.6. Para calcular `fracaso[8]`, s = `fracaso[7]` = 5. Pero $p_7 \neq p_5$, así que recalculamos s = `fracaso[5]` = 3. Sin embargo, $p_7 \neq p_3$, así que recalculamos s = `fracaso[3]` = 1. Como $p_7 \neq p_1$, s = `fracaso[1]` = 0 termina la búsqueda, y asignamos $s + 1 = 1$ a `fracaso[8]`. ■

Algoritmo 11.2 Construcción de diagramas de flujo KMP

Entradas: *P*, una cadena de caracteres; *m*, la longitud de *P*.

Salida: `fracaso`, el arreglo de ligas de fracaso, definido para los índices 1, ..., *m*. El arreglo se pasa como parámetro y el algoritmo lo llena.

```
void kmpPrep(char[] P, int m, int[] fracaso)
   int k, s;
1. fracaso[1] = 0;
2. for (k = 2; k <= m; k ++)
3.    s = fracaso[k-1];
4.    while (s >= 1)
5.       if (p[s] = p[k-1])
6.          break;
7.       s = fracaso[s];
8.    fracaso[k] = s + 1;
```

11.3.4 Análisis de la construcción de diagramas de flujo KMP

Sea *m* la longitud del patrón *P*. En esta explicación supondremos que $m \geq 2$. Es fácil ver que la complejidad del algoritmo 11.2 está en $O(m^2)$. El cuerpo del ciclo **`for`** se ejecuta $m - 1$ veces, en cada ocasión el cuerpo del ciclo **`while`** se ejecuta cuando más *m* veces porque *s* inicia en algún punto de *P* y "salta" hacia atrás, hasta cero en el peor caso. Sin embargo, este análisis no es lo bastante minucioso.

Contaremos las comparaciones de caracteres, como hicimos con el algoritmo 11.1. Puesto que la comparación de caracteres se ejecuta en cada pasada por el ciclo **`while`**, el tiempo de ejecución del algoritmo está acotado por un múltiplo del número de comparaciones de caracteres. (En realidad, puesto que la comparación no se efectúa cuando $s = 0$, cabe señalar también que la condición $s = 0$ no se puede presentar más de $m - 1$ veces.)

Decimos que una comparación *tuvo éxito* si $p_s = p_{k-1}$, y que fue *fallida* en caso contrario. Una comparación con éxito causa una salida del ciclo **`while`**, así que no se pueden efectuar más de $m - 1$ comparaciones con éxito (una para cada *k* desde 2 hasta *m*). Después de cada comparación fallida, se decrementa *s* (puesto que `fracaso[s] < s`), así que podemos acotar el número de comparaciones fallidas determinando cuántas veces puede decrementarse *s*. Observemos lo siguiente:

1. En un principio se asigna 0 a *s*, cuando $k = 2$.
2. Sólo se incrementa *s* ejecutando la línea 8 en una pasada del ciclo **`for`**, seguida de la línea 3 en la pasada subsiguiente; estos dos enunciados incrementan *s* en 1. Esto ocurre $m - 2$ veces.
3. *s* nunca es negativo.

Por lo anterior, *s* no puede decrementarse más de $m - 2$ veces. Por tanto, el número de comparaciones fallidas no puede ser mayor que $m - 2$ y el número total de comparaciones de caracteres es cuando más $2m - 3$. Obsérvese que, para contar comparaciones de caracteres, en realidad contamos el número de veces que cambia el índice *s*. Esto último es otra buena medida del trabajo efectuado por el algoritmo. La conclusión importante es que la complejidad de la construcción del diagrama de flujo es linear en términos de la longitud del patrón.

11.3.5 El algoritmo de exploración Knuth-Morris-Pratt

Ya describimos informalmente el procedimiento para usar el diagrama de flujo KMP para explorar el texto. Ahora presentamos el algoritmo.

Algoritmo 11.3 Exploración KMP

Entradas: *P* y *T*, las cadenas del patrón y el texto; *m*, la longitud de *P*; `fracaso`, el arreglo de ligas de fracaso establecido con el algoritmo 11.2. La longitud de *P* se habrá determinado al establecer el arreglo `fracaso`. Se supone que el patrón no está vacío.

Salidas: El valor devuelto es el índice en *T* en el que inicia una copia de *P*, o -1 si no se encuentra *P* en *T*.

Comentario: La función `finTexto` es la de la definición 11.1.

```
int explorKMP(char[] P, char[] T, int m, int[] fracaso)
    int coincide;
    int j, k;
    // j indiza caracteres del texto;
    // k indiza el patrón y el arreglo fracaso.
    coincide = -1;
    j = 1; k = 1;
    while (finTexto(T, j) == false)
      if (k > m)
         coincide = j - m;   // Se halló coincidencia
         break;

      if (k == 0)
         j ++;
         k = 1;   // Iniciar otra vez el patrón
      else if (T[j] == P[k])
         j ++;
         k ++;
      else
         // Seguir flecha de fracaso.
         k = fracaso[k];
      // Continuar el ciclo.
    return coincide;
```

El análisis del algoritmo de exploración utiliza un argumento muy similar al que se usó para analizar el algoritmo que establece las ligas de fracaso y se deja como ejercicio 11.8. El número de comparaciones de caracteres efectuadas por el algoritmo 11.3 es de cuando más $2n$, donde n es la longitud del texto *T*. Por tanto, el algoritmo para cotejo de patrones Knuth-Morris-Pratt, que consiste en los algoritmos 11.2 y 11.3, efectúa $\Theta(n + m)$ operaciones en el peor caso, una mejora importante respecto a la complejidad de peor caso del algoritmo 11.1, que era $\Theta(mn)$. Ciertos estudios empíricos han mostrado que los dos algoritmos efectúan aproximadamente el mismo número de comparaciones de caracteres en promedio (con textos en lenguaje natural), pero el algoritmo KMP nunca tiene que retroceder en el texto.

11.4 El algoritmo Boyer-Moore

El peor defecto de la Reina Almudena
Era masticar pedacitos de cadena.[*]

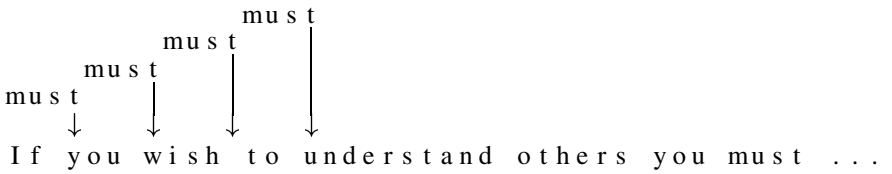
Tanto con `cotejoSimple` (algoritmo 11.1) como con `explorKMP` (algoritmo 11.3), si el patrón *P* se halla a partir de la *i*-ésima posición del texto, ello implica que ya se examinaron todos los caracteres del texto que están antes de la *i*-ésima posición (es decir, que cada uno de esos caracteres ha participado en por lo menos una comparación). La idea clave del algoritmo Boyer-Moore es que podrían pasarse por alto algunos caracteres del texto. De hecho, nuestra intuición sugiere que, cuanto más largo sea el patrón, más información tendrá el algoritmo acerca de lo que debe encontrar, así que un buen algoritmo debería poder saltar con mayor rapidez los lugares del texto en los que el patrón no puede aparecer.

11.4.1 La idea nueva

El algoritmo Boyer-Moore (BM de aquí en adelante) siempre explora el patrón de derecha a izquierda y utiliza dos heurísticas para decidir qué distancia puede deslizarse el patrón sobre la cadena de texto después de hallar una diferencia. Como es costumbre, sea *P* un patrón de longitud *m* y *T* una cadena de texto de longitud *n*. Ilustramos la primera heurística para deslizar *P* con el ejemplo siguiente.

Ejemplo 11.5 Primera heurística de Boyer-Moore

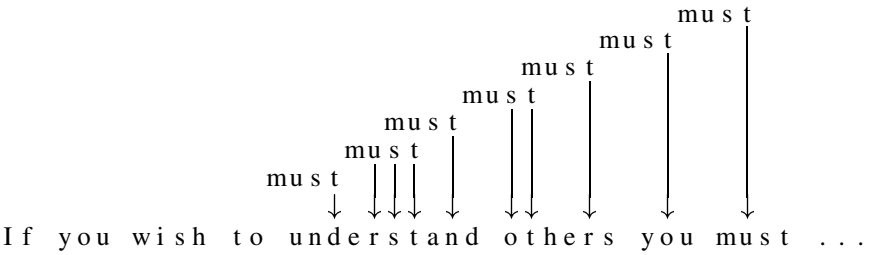
Estamos buscando el patrón *must* en una cita de Oscar Wilde: "If you wish to understand others you must intensify your own individualism." ("Si quieres entender a los demás, debes intensificar tu propio individualismo.") El patrón se ubica sobre la cadena en cada lugar en el que se verificará una posible coincidencia. Las flechas indican comparaciones. Las primeras cuatro comparaciones son:



Cuando se compara el último carácter de *must*, la '*t*', con la '*y*' de *you*, observamos no sólo que no hay coincidencia en esta posición, sino también que, dado que no hay ninguna '*y*' en *must*, no puede haber ninguna coincidencia que se traslape con la '*y*'. Podríamos deslizar el patrón cuatro posiciones a la derecha. Asimismo, después de cada una de las dos comparaciones siguientes, deslizaremos el patrón cuatro posiciones porque no hay una '*w*' ni un espacio en blanco en *must*. En la cuarta comparación tampoco hay coincidencia, pero sí hay una '*u*' en el patrón, así que deslizamos el patrón de modo que las dos '*u*' estén alineadas, como se muestra en la primera comparación del diagrama siguiente, y se verifica si hay coincidencia o no. (Como siempre en BM,

* El traductor pide perdón a Hillaire Belloc por alterar la cita tomada de su obra *Cuentos con moraleja* (1907).

comenzamos a verificar en el extremo derecho del patrón.) Las comparaciones subsiguientes son:



Al detectarse la diferencia entre la '*u*' de *must* y la '*r*' de *understand*, el patrón se desliza apenas lo suficiente para rebasar la '*r*'. Lo mismo sucede con la '*s*' de *must* y la '*o*' de *others*. La última comparación revela una diferencia, pero el carácter del texto '*u*' sí aparece en el patrón, así que éste se desliza hasta alinear las '*u*'. Cuatro comparaciones más (de derecha a izquierda) confirmarán que se ha hallado una coincidencia perfecta. ■

En este ejemplo sólo se efectuaron 18 comparaciones de caracteres, pero en vista de que la coincidencia ocurre en la posición 38 de *T*, los otros algoritmos efectuarían por lo menos 41 comparaciones. Sin embargo, a diferencia de KMP, este algoritmo debe poder retroceder en el texto una distancia igual a la longitud del patrón.

El número de posiciones que podemos "saltar" hacia adelante al detectar una diferencia dependerá del siguiente carácter que se leerá, digamos t_j. Almacenaremos estos números en un arreglo llamado `saltoCar` indizado por el conjunto de caracteres Σ.

Detalle de Java: Los caracteres (tipo **`char`**) en Java ocupan 16 bits, así que un arreglo con un elemento para cada carácter tendría 65,536 celdas. El tipo **`byte`** sólo ocupa 8 bits y requiere un arreglo de 256 celdas. Si se sabe (o se supone) que el texto y el patrón sólo contienen caracteres de 8 bits (lo cual incluye todos los caracteres de la generalidad de los teclados), se podría usar el arreglo más pequeño.

Para controlar el algoritmo de exploración, es más útil conocer la cantidad en la que deberá incrementarse el índice del texto *j* para iniciar la siguiente exploración de derecha a izquierda del patrón, en lugar de la distancia que *P* se deslizará hacia adelante. Como puede verse en el ejemplo 11.5 y la figura 11.7, este salto en *j* podría ser mayor que la distancia que *P* se desliza. Si t_j no aparece en *P*, podremos saltar hacia adelante *m* posiciones. La figura 11.7 ilustra la forma de calcular el salto en el caso en que t_j sí aparece en *P*. De hecho, t_j podría aparecer más de una vez en *P*. Necesitamos efectuar el salto más pequeño posible, alineando el ejemplar de t_j que aparece más a la derecha en *P*; si no lo hacemos así, podríamos pasar por alto una copia de *P*. (Nunca conviene deslizar el patrón hacia atrás; si nuestra posición actual en *P* ya está a la izquierda del ejemplar de t_j que está más a la derecha en *P*, `saltoCar`[t_j] no será útil.)

Algoritmo 11.4 Cálculo de saltos para el algoritmo Boyer-Moore

Entradas: La cadena patrón *P*; *m*, la longitud de *P*; el tamaño del alfabeto, `alfa` = $|\Sigma|$.

Salidas: El arreglo `saltaCar`, definido para los índices 0, . . . , `alfa`−1. El arreglo se pasa como parámetro y el algoritmo lo llena.

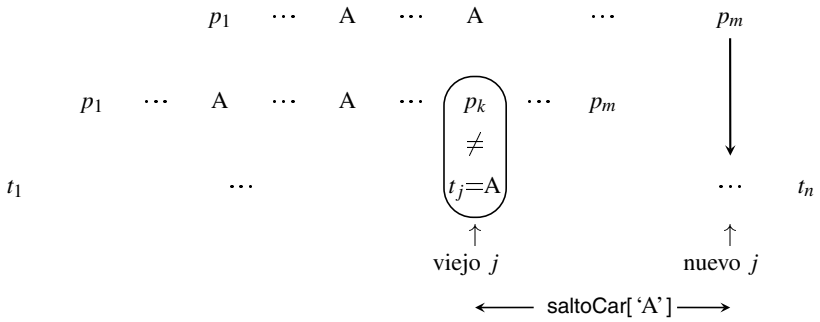


Figura 11.7 Deslizamiento del patrón para alinear un carácter coincidente

```
void calcularSaltos(char[] P, int m, int alfa, int[] saltoCar)
    char car;
    int k;
    for (car = 0; car < alfa; car ++)
        saltoCar[car] = m;
    for (k = 1; k <= m; k ++)
        saltoCar[P[k]] = m − k;
```

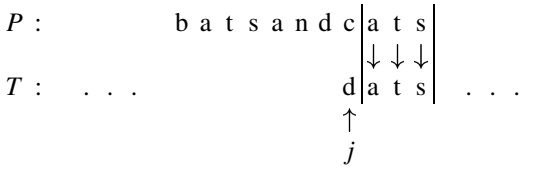
Es evidente que el tiempo requerido para calcular los saltos está en $\Theta(|\Sigma| + m)$, donde m es la longitud del patrón P.

11.4.2 Y la idea "vieja"

El simple uso de `saltoCar` para avanzar a saltos por el texto hace que el algoritmo Boyer-Moore se ejecute con mucha mayor rapidez que el algoritmo Knuth-Morris-Pratt en muchos casos. La combinación de `saltoCar` con una idea similar a la de las flechas `fracaso` del algoritmo KMP puede mejorar aún más el algoritmo.

Ejemplo 11.6 Segunda heurística de Boyer-Moore

Supóngase que algún segmento (de extrema derecha) de P coincidió con una parte de T antes de hallarse una diferencia.



El carácter de texto actual es una '*d*'. Utilizando `saltoCar['d']`, deslizaríamos el patrón sólo una posición a la derecha para alinear su '*d*' con la '*d*' de T. Sin embargo, sabemos que las letras de T que están a la derecha de la posición actual son '*ats*', las mismas letras que forman el

sufijo de P recién examinado. Si sabemos que P no tiene otro ejemplar de '*ats*', podremos deslizar P hasta haber rebasado '*ats*' en T. Si P sí tiene un ejemplar previo de '*ats*', podríamos deslizar P de modo que el '*ats*' previo quede alineado con las letras coincidentes de T. En el ejemplo anterior, la siguiente posición para una posible coincidencia es:

```
P :          b|a t s|a n d c a t s
T : . . .    d|a t s|  . . .
                                ↑
                             nuevo j
```

Para no omitir una posible coincidencia, si P tiene más de una subcadena que coincida con el sufijo coincidente, alinearemos la que esté más a la derecha (sin contar, claro, el sufijo mismo). ■

En la figura 11.8(a) se muestra el panorama general; la diferencia se detecta en p_k y t_j. La figura 11.8(b) muestra el patrón deslizado hacia la derecha para alinear una subcadena con el sufijo coincidente. La figura 11.8(c) muestra otra posibilidad, en la que el sufijo coincidente no se repite en alguna otra parte del patrón. En tal caso se busca un prefijo que coincida con *algún* sufijo, luego el patrón se desliza a la derecha para alinearlos. Queremos que `saltoCotejo[k]` sea la cantidad en que se incrementa j, el índice de la posición en el texto, para iniciar la siguiente exploración de derecha a izquierda del patrón una vez que se ha detectado una diferencia en p_k.

Definición 11.4 `saltoCotejo` y `deslizar`

Para $1 \leq k \leq m$, definimos

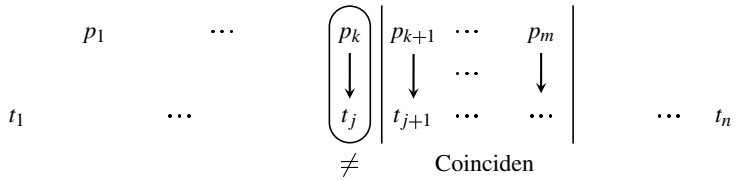
$$\texttt{saltoCotejo[k]} = \texttt{deslizar[k]} + m - k \tag{11.1}$$

y definimos `deslizar[k]` como se describe a continuación. Intuitivamente, `deslizar[k]` es la distancia que podemos deslizar el patrón hacia adelante después de detectarse una diferencia en p_k, y $m - k$ es el número de caracteres que coincidieron antes de encontrarse una diferencia. Su suma es la distancia que puede saltar el índice del texto, j.

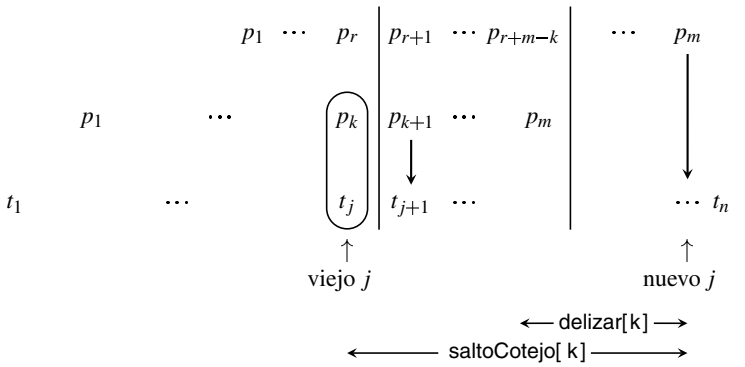
Sea r el índice más grande tal que $p_{k+1} \cdots p_m$ coincide con $p_{r+1} \cdots p_{r+m-k}$ y $p_k \neq p_r$. (Obsérvese que $r < k$.) Es decir, el *sufijo* de $m - k$ caracteres de P es igual a la subcadena de $m - k$ caracteres que principia en el índice $r + 1$, la coincidencia no se puede extender hacia la izquierda, ya que r es el índice más grande para el cual esto se cumple. Entonces, `deslizar[k] = k - r`. Si desplazamos el patrón esta distancia hacia adelante, la subcadena que principia en $r + 1$ quedará alineada con el texto en el lugar en el que solía estar alineada la subcadena que principia en $k + 1$.

Incluimos la condición $p_r \neq p_k$ porque, si vamos a usar `saltoCotejo[k]` en este punto, ya sabemos que p_k no coincide con t_j. Si $p_r = p_k$, entonces p_r tampoco coincidirá con t_j, así que no tiene caso alinearlos. Si $k = m$, el sufijo $p_{k+1} \cdots p_m$ está vacío, por lo que se satisface el requisito de coincidencia con cualquier opción de r. En este caso, r es el índice más grande tal que $p_r \neq p_k$.

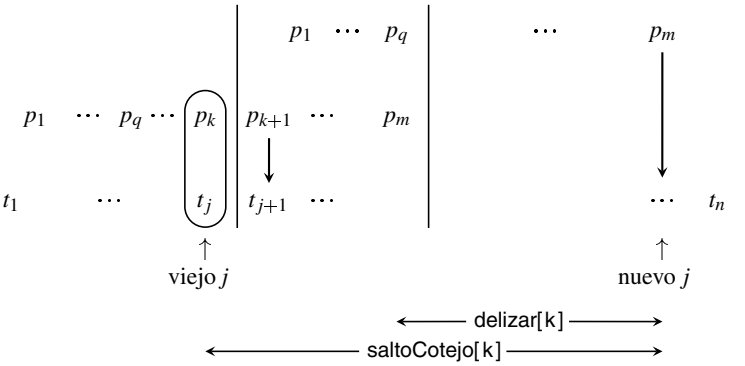
A veces no existe en P una subcadena que coincida con el sufijo coincidente, $p_{k+1} \cdots p_m$. En tal caso alineamos el *prefijo* más grande de P que coincida con algún sufijo de P. Si este prefijo tiene q caracteres, entonces `deslizar[k] = m - q`. ■



(a) Hay una diferencia en p_k y t_j.



(b) Alineamos la subcadena de P que coincida con el sufijo coincidente y esté más a la derecha (y que satisfaga $p_r \neq p_k$).

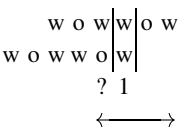


(c) Si ninguna subcadena de P coincide con $p_{k+1} \cdots p_m$, alineamos un prefijo que coincida parcialmente.

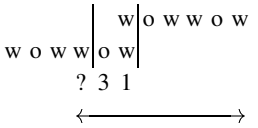
Figura 11.8 Deslizamiento del patrón para alinear una subcadena coincidente

Ejemplo 11.7 Cálculo de `deslizar` y `saltoCotejo`

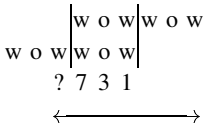
Sea P = '*wowwow*', con $m = 6$. Calcularemos los valores de `deslizar` y `saltoCotejo` a partir del extremo derecho del patrón. Abajo del patrón se muestran los valores de `saltoCotejo` que ya se calcularon. El signo de interrogación indica la posición con la que estamos trabajando ahora. En cada paso deslizamos el patrón hacia la derecha para alinear una subcadena que coincide



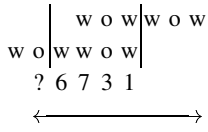
(a) Coinciden = 1, deslizar = 2, salto = 3.



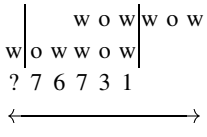
(b) Coinciden = 2, deslizar = 5, salto = 7. Obsérvese que no se usó la coincidencia del primer '*ow*' y el segundo '*ow*' porque ambos van precedidos por una '*w*'; si se detecta una diferencia en la posición 4 del patrón, no tiene caso alinear otra '*w*' en esa posición al explorar el texto.



(c) Coinciden = 3, deslizar = 3, salto = 6.



(d) Coinciden = 4, deslizar = 3, salto = 7.



(e) Coinciden = 5, deslizar = 3, salto = 8.

	P:	w	o	w	w	o	w
`saltoCotejo:`		8	7	6	7	3	1

(f) Valores finales

Figura 11.9 Cálculo de `saltoCotejo`— un ejemplo: *coinciden* es el número de caracteres que coincidieron antes de la diferencia, como en la definición 11.4.

con un sufijo. El carácter que precede a la subcadena debe ser diferente del carácter que precede al sufijo. Para la posición de la extrema derecha, asignamos 1 a `saltoCotejo` porque $p_5 \neq p_6$.

Véase la figura 11.9. Cabe señalar que este ejemplo ilustra los valores de `saltoCotejo`, pero no los pasos reales que ejecuta el algoritmo siguiente para calcularlos. Volveremos a este ejemplo después de presentar el algoritmo. ■

Algoritmo 11.5 Cálculo de saltos con base en coincidencias parciales

Entradas: *P*, la cadena patrón; *m*, la longitud de *P*.

Salidas: Un arreglo `saltoConejo`, definido para los índices 1, ..., *m*. El arreglo se pasa como parámetro y el algoritmo lo llena.

Comentario: La inicialización y las dos primeras fases en realidad calculan `deslizar[k]`, que se describió en la definición 11.4, pero lo almacenan en `saltoCotejo[k]`. La última fase convierte los elementos de `deslizar` a `saltoCotejo` según la ecuación (11.1). El arreglo `sufijo` es un análogo de derecha a izquierda del arreglo `fracaso` de KMP. Si `sufijo[k] = x`, ello quiere decir que la subcadena $p_{k+1} \cdots p_{k+m-x}$ coincide con el sufijo $p_{x+1} \cdots p_m$. Cabe señalar que `sufijo[0]` nos dice qué sufijo coincide con un prefijo de *P*.

```
void calcularSaltosCotejo(char[] P, int m, int[] saltoCotejo)
    int k, r, s, bajo, desplazar;
    int[] sufijo = new int[m+1];

    for (k = 1; k <= m; k ++)
       saltoCotejo[k] = m + 1;  // Deslizamiento imposiblemente grande

    // Calcular ligas sufijo (como ligas fracaso de KMP, pero dere-
    cha a izq.).
    // Detectar si subcadena es igual a sufijo coincidente y está
    // precedida por diferencia en s; calcular su deslizamiento.
    sufijo[m] = m + 1;
    for (k = m-1; k >= 0; k --)
       s = sufijo[k+1];
       while (s <= m)
          if (P[k+1] == P[s])
             break;   // Salir del ciclo while.
          // Diferencia entre k+1 y s.
          saltoCotejo[s] = min(saltoCotejo[s], s - (k+1));
          s = sufijo[s];
       sufijo[k] = s - 1;
       // Continuar ciclo for.

    // Si no hay coincidencia de sufijo en k+1, calcular el desli-
       zamiento con base en un prefijo que coincida con el sufijo.
    // Longitud prefijo = (m - desplazar).
    bajo = 1; desplazar = sufijo[0];
    while (desplazar <= m)
       for (k = bajo; k <= desplazar; k ++)
          saltoCotejo[k] = min(saltoCotejo[k], desplazar);
       bajo = desplazar + 1; desplazar = sufijo[desplazar];

    // Sumar número de caracteres coincidentes a distancia de
    // deslizamiento.
    for (k = 1; k <= m; k ++)
       saltoCotejo[k] += (m - k);
    return;
```

Ejemplo 11.8 Cálculo de `deslizar` y `saltoCotejo` con el algoritmo

Consideremos el patrón '*wowwow*' que usamos en el ejemplo 11.7. En la primera fase, `sufijo` y `saltoCotejo` (que lógicamente es `deslizar`, véase la definición 11.4) reciben los valores siguientes (los espacios en blanco indican que no se ha calculado un valor, así que persiste el valor inicial imposiblemente grande).

P		w	o	w	w	o	w
sufijo	3	4	5	5	6	6	7
saltoCotejo						2	1

En la segunda fase, dado que `sufijo[0] = 3`, se asigna el valor 3 a las posiciones de la 1 a la 3 de `saltoCotejo`. Luego se recupera `sufijo[3]` y se ve que es 5. Se asigna el valor 5 a `saltoCotejo[4]`. Sin embargo, `saltoCotejo[5]` ya tiene un valor (que necesariamente es menor que 5), por lo que no se modifica. Asimismo, `saltoCotejo[6]` conserva su valor anterior. En este punto tenemos los valores finales de `deslizar` (almacenados en `saltoCotejo` para ahorrar espacio).

P	w	o	w	w	o	w
`saltoCotejo`	3	3	3	5	2	1

Por último, se suma el número de caracteres que habían coincidido antes de detectarse la diferencia en k, para convertir `deslizar` en `saltoCotejo`, según la ecuación (11.1).

P	w	o	w	w	o	w
`saltoCotejo`	8	7	6	7	3	1

Estos valores coinciden con los que se calcularon por inspección en el ejemplo 11.7. ■

La base para entender la corrección del algoritmo 11.5 es que el arreglo `sufijo` es un análogo de derecha a izquierda del arreglo `fracaso`. Por tanto, `sufijo[k] > k` se cumple en lugar de `fracaso[k] < k`, `sufijo[m] = m + 1` se cumple en lugar de `fracaso[1] = 0`, y así. La propiedad importante es que `sufijo[k] = x` si y sólo si la subcadena $p_{k+1} \cdots p_{k+m-x}$ coincide con el sufijo $p_{x+1} \cdots p_m$. La demostración de que el cálculo hace que se tenga esta propiedad es similar a la demostración correspondiente para las ligas de fracaso KMP, que se presentó informalmente en la sección 11.3.3.

Consideremos ahora la sucesión de índices definida por $r_0 =$ `sufijo[0]`, $r_{i+1} =$ `sufijo[`r_i`]`, para $i \geq 0$ hasta $r_{i+1} = m + 1$. Éstos son los valores que adopta la variable `desplazar`. Por la propiedad anterior del arreglo `sufijo`, se sigue que todo sufijo del patrón que principia en $r_i + 1$ (y que termina en m, desde luego) es también un prefijo de P.

Faltan aún muchos detalles para demostrar que el algoritmo es correcto, pero los puntos anteriores cubren las ideas principales. Si desea conocer más detalles, remítase a las fuentes que se mencionan en Notas y referencias al final del capítulo.

Análisis del cálculo de saltos de cotejo

El tiempo para la primera fase de `calcularSaltosCotejo`, en la que se calcula el arreglo `sufijo`, se puede analizar con argumentos similares a los que se usaron para analizar el cálculo de las ligas de fracaso KMP en la sección 11.3.4. Dicho tiempo está en $\Theta(m)$. Es fácil ver que el resto del algoritmo también está en $\Theta(m)$ y también que usa $\Theta(m)$ espacio extra para el arreglo `sufijo`.

11.4.3 El algoritmo de exploración Boyer-Moore

Algoritmo 11.6 Exploración Boyer-Moore

Entradas: P y T, las cadenas de patrón y de texto; m, la longitud de P; `saltoCar` y `saltoCotejo`, los arreglos descritos en las secciones 11.4.1 y 11.4.2. Se supone que el patrón no está vacío.

Salidas: El valor devuelto es el índice de T en el que principia una copia de P, o -1 si no se encuentra P en T.

Comentario: La función `finTexto` se ajusta a la definición 11.1.

```
int explorBM(char[] P, char[] T, int m, int[] saltoCar, int[] salto-
Cotejo)
    int coincide;
    int j, k;
    // j indiza caracteres del texto;
    // k indiza el patrón.
    coincide = -1;
    j = m; k = m;
    while (finTexto(T, j) == false)
       if (k < 1)
          coincide = j + 1;   // Se halló coincidencia
          break;

       if (T[j] == P[k])
          j --; k --;
       else
          // deslizar P hacia adelante
          j += max(saltoCar[T[j]], saltoCotejo[k]);
          k = m;
       // Continuar ciclo.
    return coincide;
```

11.4.4 Comentarios

El comportamiento del algoritmo Boyer-Moore depende del tamaño del alfabeto y de la repetición dentro de las cadenas. En estudios empíricos empleando textos en lenguaje natural y $m \geq 5$, el algoritmo efectuó apenas de 0.24 a 0.3 comparaciones de caracteres por carácter del texto, hasta el punto en que se halló el patrón o se terminó el texto. Dicho de otro modo, sólo se examina entre una cuarta parte y un tercio de los caracteres. En la figura 11.10 se presentan los resultados de uno de esos estudios en el que se compararon los tres algoritmos. En los experimentos se usaron 20 patrones con longitud desde $m = 1$ hasta $m = 14$. La longitud del texto fue 5000.

En el caso de cadenas binarias, BM no tiene tan buen desempeño (`saltoCar` no ayuda mucho); en otro estudio, se efectuaron aproximadamente 0.7 comparaciones por cada carácter del texto.

En todos los casos con $m \geq 5$, el número medio de comparaciones está acotado por cn, para una constante $c < 1$. Si el patrón es muy pequeño ($m \leq 3$), no vale la pena incurrir en el trabajo extra de someter el patrón a un procesamiento previo; BM efectúa más comparaciones que el enfoque directo, `cotejoSimple` (algoritmo 11.1).

Existen varias mejoras y modificaciones del algoritmo BM que hacen que se ejecute en menos tiempo. (Véanse las Notas y referencias al final del capítulo.) Al igual que el algoritmo 11.1, BM retrocede en la cadena de texto (porque explora el patrón de derecha a izquierda).

Suelen ser útiles dos extensiones del problema de cotejo de patrones: hallar *todas* las ocurrencias del patrón en el texto, y hallar cualquiera de un conjunto finito de patrones en el texto.

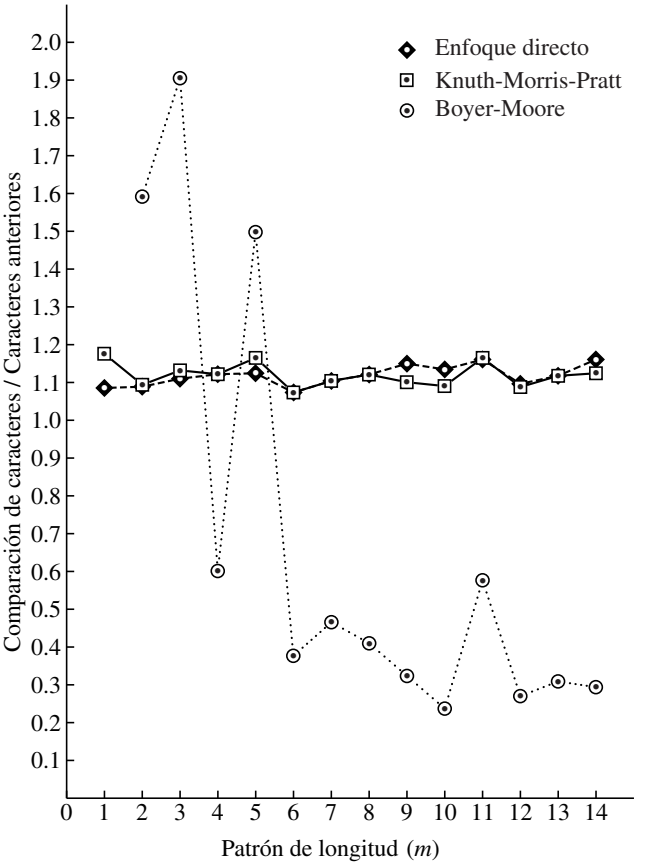


Figura 11.10 Comparación de algoritmos para cotejar cadenas. (De G. de V. Smit, "A Comparison of Three String Matching Algorithms", *Software: Practice and Experience*, vol. 12, Derechos Reservados 1982, John Wiley and Sons, Ltd. Reproducción autorizada por John Wiley and Sons, Ltd.)

11.5 Cotejo aproximado de cadenas

En las secciones 11.2 a 11.4 estudiamos varios algoritmos para hallar una copia de una cadena de caracteres a la que llamamos *patrón* en otra cadena a la que llamamos *texto*. Esos algoritmos buscaban una copia exacta del patrón. Sin embargo, en muchas aplicaciones no se espera hallar una copia exacta. Un corrector de ortografía, por ejemplo, podría buscar en un diccionario una entrada similar a una palabra dada (mal escrita). En el reconocimiento del habla, las muestras podrían variar. Otras aplicaciones en las que se buscan coincidencias cercanas, pero no exactas, van desde la identificación de sucesiones de aminoácidos hasta el reconocimiento de gorjeos de pájaros. Al igual que en las secciones anteriores, aquí usaremos cadenas de caracteres, pero es eviden-

te que el método puede servir con cadenas de "alfabetos" que consisten en otros tipos de datos, lo cual sería el caso en, por ejemplo, el reconocimiento del habla.

En esta sección presentaremos una solución de programación dinámica para el problema de hallar una coincidencia aproximada de un patrón en una cadena. El paradigma de la programación dinámica se introdujo en el capítulo 10. Los conocimientos de programación dinámica ayudarán al lector a entender mejor este problema.

Sea $P = p_1 p_2 \cdots p_m$ el patrón y $T = t_1 t_2 \cdots t_n$ el texto. Supondremos que n es grande en comparación con m. La terminología que introducimos a continuación adopta el punto de vista de que P es un patrón "correcto", mientras que T podría ser sólo aproximado. En muchas aplicaciones, T podría tener "ruido". No obstante, el algoritmo no depende de que se adopte dicho punto de vista.

Definición 11.5 Coincidencia k-aproximada

Sea k un entero no negativo. Una *coincidencia k-aproximada* es una coincidencia de P en T que no tiene más de k diferencias. Las diferencias pueden ser de cualquiera de los tres tipos siguientes. El nombre de la diferencia es la operación que se tendría que ejecutar en T para acercarlo a P.

modificar:	Los caracteres correspondientes en P y T son distintos.
borrar:	T contiene un carácter que no está en P.
insertar:	T no tiene un carácter que aparece en P. ■

Ejemplo 11.9

La coincidencia que se muestra a continuación es 3-aproximada, e incluye una de cada una de las diferencias permitidas. No hay espacios en blanco en P ni en T; los espacios se usan para mostrar la coincidencia más claramente.

```
P :          u n n e   c e s s a r i l y
                ↓ ↓         ↓
T : . . . un escessaraly . . .
```

■

Las entradas del problema son P, T, m (la longitud de P) y k (el número aceptable de diferencias). El problema consiste en hallar una subcadena de T que coincida k-aproximadamente con P, o determinar que no hay ninguna coincidencia k-aproximada.

En el problema de cotejo exacto, si el carácter actual del texto no coincide con el patrón, sólo hay una acción posible: desplazar el patrón. En este problema hay cuatro opciones (a menos que se exceda k): desplazar el patrón o ejecutar una de las cuatro operaciones de la definición 11.5. No hay alguna manera obvia de saber cuál opción será la mejor, así que el enfoque directo consiste en desarrollar un procedimiento recursivo para evaluar las alternativas. En el capítulo 10 vimos varios problemas de optimización que podrían resolverse dentro del marco siguiente:

1. Para cada opción actual:
   a. Determinar qué subproblema(s) quedarían si se eligiera esta opción.
   b. Determinar recursivamente los costos óptimos de esos subproblemas.
   c. Combinar esos costos con el costo de la opción actual para obtener el costo total de esta opción.
2. Seleccionar la opción actual que tenga el costo total más bajo.

Con un poco de experimentación veremos que este procedimiento se topa con muchos subproblemas repetidos.

Ejemplo 11.10

Supóngase que en el texto se transponen dos letras del patrón:

P:		A B C D E	
T:	. . .	A C B D E	. . .

Suponiendo que el patrón se explora del izquierda a derecha, esas diferencias se pueden explicar como dos *modificar*, o como *borrar*(C) seguido más adelante por *insertar*(C), o como *insertar*(B) seguido más adelante por *borrar*(B). En los tres casos, falta por resolver el subproblema de cotejar 'DE'. Una búsqueda con retroceso lo resolvería de nuevo en cada rama de la búsqueda. Éste es exactamente el tipo de comportamiento que indica una posible aplicabilidad del paradigma de la programación dinámica. ■

Para preparar una solución de programación dinámica necesitamos formalizar la solución recursiva de retroceso y decidir cómo identificaremos los subproblemas. Podríamos querer explorar el patrón de izquierda a derecha, como en KMP, o de derecha a izquierda, como en BM. (Supondremos que el texto generalmente se procesa de izquierda a derecha en ambos casos, aunque podría haber algo de retroceso.)

Para la exploración del patrón de izquierda a derecha, una forma natural de identificar un subproblema es con un par de enteros (i, j), donde i denota el principio de un sufijo del patrón y j denota la posición del texto en la que se debe iniciar el cotejo. Entonces, el subproblema $(i\ j)$ se especifica como hallar la coincidencia con diferencia mínima entre $p_i \cdots p_m$ y un segmento de T que principia en t_j. Esto podría introducir algunas complicaciones si no resulta práctico mirar muy hacia adelante en el texto. Cabe señalar que, en las ramas de la búsqueda con retroceso en las que se escoge *borrar*, (i, j) depende de $(i, j + 1)$, que depende de $(i, j + 2)$, y así sucesivamente. Nos veremos obligados a mirar hacia adelante en el texto sin avanzar en el patrón. Veamos si la alternativa, una exploración de derecha a izquierda del patrón, parece más sencilla.

Para la exploración del patrón de derecha a izquierda, una forma natural de identificar un subproblema es con un par de enteros (i, j), donde i denota el *final* de un *prefijo* del patrón, y j denota la posición en el texto en la que el cotejo debe *terminar*. Ahora el subproblema (i, j) se especifica como hallar la coincidencia de diferencia mínima entre $p_1 \cdots p_i$ y un segmento de T que *termina* en t_j. Ahora, en las ramas de la búsqueda con retroceso en las que se escoge *borrar*, $(i\ j)$ depende de $(i, j - 1)$, que depende de $(i, j - 2)$, y así sucesivamente. Sin embargo, si nuestra es-

trategia general para el algoritmo de programación dinámica consiste en resolver el problema en orden creciente de j, estos subproblemas ya se habrán encontrado y resuelto antes. Por tanto, la exploración del patrón de derecha a izquierda parece ser una mejor base para construir una solución de programación dinámica.

Para resolver recursivamente el subproblema (i, j), necesitamos resolver estos subproblemas: $(i, j - 1)$, debido a *borrar*; $(i - 1, j)$, debido a *insertar*; e $(i - 1, j - 1)$, debido a *modificar* o a que los caracteres coinciden. Sin embargo, si resolvemos los subproblemas "de abajo hacia arriba" en un orden tal que los últimos tres subproblemas mencionados se resuelvan antes de iniciarse la resolución de (i, j), podremos resolver (i, j) consultando las soluciones anteriores. Para la solución de programación dinámica, definimos:

Definición 11.6 Tabla de diferencias

$D[i][j]$ = el número mínimo de diferencias entre $p_1 \cdots p_i$ y un segmento de T que termina en t_j. ■

Habrá una coincidencia k-aproximada que termina en t_j para cualquier j tal que $D[m][j] \leq k$. Por tanto, si queremos hallar la primera coincidencia k-aproximada, podremos detenernos tan pronto como hallemos un elemento menor o igual que k en la última fila de D. Las reglas para calcular los elementos de D consideran cada una de las posibles diferencias que podrían presentarse entre p_i y t_j y, desde luego, la posibilidad de que esos dos caracteres sean iguales. $D[i][j]$ es el mínimo de los cuatro valores siguientes:

$$\begin{aligned}
\texttt{costoCotejo} &= D[i-1][j-1] && \text{si } p_i = t_j \\
\texttt{costoModif} &= D[i-1][j-1] + 1 && \text{si } p_i \neq t_j \\
\texttt{costoInsertar} &= D[i-1][j] + 1 \\
\texttt{costoBorrar} &= D[i][j-1] + 1
\end{aligned}$$

Cada elemento sólo necesita elementos arriba de él y a su izquierda en la tabla (véase la figura 11.11), así que el cálculo se puede efectuar en orden hacia adelante fila por fila o columna por columna. Puesto que n podría ser mucho mayor que m, resulta mucho más eficiente calcular los elementos de D columna por columna. Para iniciar el cálculo, utilizamos una fila 0, con $D[0][j] = 0$ para todo j (intuitivamente, dado que una sección nula del patrón difiere en cero posiciones res-

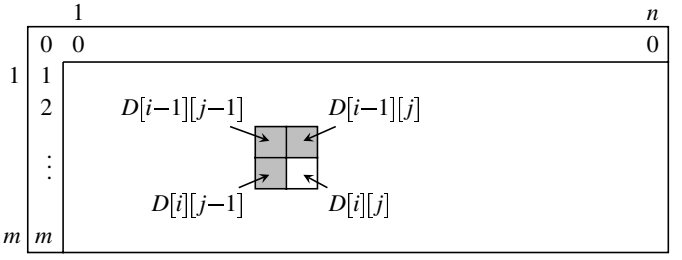


Figura 11.11 Cálculo de $D[i][j]$: se usan los tres elementos sombreados

		H	a	v	e		a		h	s	p	p	y		d	a	y	.
	0	0	0	0	0	0	0	0	0	0	0							
h	1	1	1	1	1	1	1	1	0	1	1	1	1					
a	2	2	1	2	2	2	1	2	1	1	2	2	2					
p	3	3	2	2	3	3	2	2	2	2	1	2	3					
p	4	4	3	3	3	4	3	3	3	3	2	1	2					
y	5	5	4	4	4	4	4	4	4	4	3	2	1					

Tabla 11.2 La tabla *D* del ejemplo 11.11

pecto al sufijo nulo de $t_1 \cdots t_j$), y una columna 0, con $D[i][0] = i$ (porque $p_1 \cdots p_i$ difiere en i posiciones respecto a un prefijo nulo de T).

Ejemplo 11.11 Cálculo de una tabla *D*

Supóngase que $P =$ '`happy`' y que T es la oración '`Have a hsppy day`' que tiene un error de ortografía. La tabla 11.2 muestra los valores de D. Los elementos se calculan columna por columna y, tan pronto como se detecta que un elemento de la quinta fila tiene el valor 1, el cálculo termina. ■

El trabajo que se efectúa para calcular cada elemento de D es una constante pequeña, por lo que el trabajo total realizado está en $O(nm)$. Esta rapidez es similar (por un factor constante) a la del primer algoritmo directo para cotejo exacto de patrones (algoritmo 11.1).

¿Qué hay con el espacio? El espacio utilizado por un algoritmo de programación dinámica para su tabla suele ser un precio razonable por el ahorro de tiempo. Sin embargo, la tabla D en este algoritmo es de m por n, y n es muy grande. Es evidente que no hay necesidad de almacenar toda la tabla. Sólo se necesitan elementos de la columna actual y de la anterior, así que el algoritmo se puede escribir empleando aproximadamente $2m$ celdas.

Escribir el algoritmo deberá ser un ejercicio fácil (véase el ejercicio 11.22). En las Notas y referencias al final del capítulo se menciona un algoritmo de cotejo k-aproximado que se ejecuta en tiempo $O(kn)$.

Ejercicios

Sección 11.2 Una solución directa

11.1 Reescriba el algoritmo 11.1 eliminando la variable *i*.

11.2 Reescriba el algoritmo 11.1 de modo que opere con entradas que son listas ligadas. Por sencillez, suponga que el tipo de los elementos de las listas es **`int`**. Utilice las operaciones del tipo de datos abstracto `ListaInt` de la sección 2.3.2 y suponga que T y P son objetos de esta clase.

11.3 En este ejercicio, diseñará operaciones para un tipo de datos abstracto `Texto` que realizarán lo que `cotejoSimple` tiene que hacer con T, sin suponer forzosamente que T es un arreglo. Trate de que sean lo bastante generales como para que otros algoritmos de cotejo de cadenas pue-

dan usarlo en muchos casos. Ya supusimos que se cuenta con `finTexto`. Otros nombres que podrían usarse para las operaciones son `adelantarTexto`, `retrocederTexto` y `obtCarTexto`.

a. Escriba las especificaciones, pero no implemente las operaciones.

b. Muestre cómo modificaría `cotejoSimple` para usar sus operaciones con *T*.

11.4 Suponga que el texto *T* es demasiado grande como para almacenarlo todo en la memoria, así que se lee conforme se va necesitando. El algoritmo 11.1 podría necesitar aproximadamente *m* caracteres previos de *T*, a la izquierda de t_j. Es decir, podría ser necesario retroceder aproximadamente *m* posiciones.

En este ejercicio, diseñará una modificación del TDA Cola, llamada `ColaAbierta`, que conserve elementos (caracteres en este caso) en orden FIFO, como hace una cola normal, pero que permita acceso a cualquier elemento de la cola, no sólo al frontal.

a. Escriba las especificaciones de sus operaciones nuevas.

b. Muestre cómo implementar de manera eficiente las operaciones necesarias utilizando un arreglo, siguiendo las sugerencias del ejercicio 2.16.

c. Bosqueje la forma de implementar el TDA `Texto` del ejercicio 11.3 empleando el TDA `ColaAbierta`. Lo que se busca es mantener suficientes caracteres en la cola como para dar cabida a cualquier retroceso que el algoritmo pueda requerir. Suponga que en el momento de crearse el objeto `ColaAbierta` el algoritmo conoce la distancia máxima que podría tener que retroceder en algún momento dado.

Sección 11.3 El algoritmo Knuth-Morris-Pratt

11.5 Dibuje el autómata finito (diagrama de flujo) para el patrón '*ABAABA*' donde $\Sigma = \{A, B, C\}$.

11.6 Dé los índices de fracaso que emplea el algoritmo KMP con los patrones siguientes:

a. *AAAB*

b. *AABAACAABABA*

c. *ABRACADABRA*

d. *ASTRACASTRA*

11.7 Dé el patrón que comienza con una '*A*' y sólo usa letras de $\{A, B, C\}$ que tendría los índices de fracaso siguientes (para el algoritmo KMP):

0 1 1 2 3 4 2 2

11.8 Demuestre que `explorKMP` (algoritmo 11.3) efectúa cuando más $2n$ comparaciones de caracteres.

11.9 ¿Cómo se comportan los algoritmos KMP si el patrón y/o el texto son nulos (tienen longitud cero)?, ¿se "congelan"?, si no lo hacen, ¿su salida es lógica y correcta?

11.10 Recuerde que el patrón $P =$ '$A \cdots AB$' ($m - 1$ *A*es seguido de una *B*) y la cadena de texto $T =$ '$A \cdots A$' (n *A*es) son una entrada de peor caso para el algoritmo 11.1.

a. Dé los valores de los índices de `fracaso` para *P*. ¿Cuántas comparaciones efectúa exactamente el algoritmo 11.2 para calcularlos?

b. ¿Cuántas comparaciones de caracteres efectúa exactamente `explorKMP` para explorar T en busca de una ocurrencia de P?

c. Dado un m arbitrariamente grande, hallar un patrón Q con m letras tal que `kmpPrep` efectúe más comparaciones de caracteres con Q que con el patrón P de m letras descrito al principio del ejercicio.

11.11 Demuestre que el algoritmo 11.2 establece las ligas de fracaso KMP de modo tal que `fracaso[k]` es el r más grande (con $r < k$) tal que $p_1 \cdots p_{r-1}$ coincide con $p_{k-r+1} \cdots p_{k-1}$.

⋆ **11.12** La estrategia para establecer las ligas de fracaso del algoritmo KMP tiene un defecto que se ilustra en la figura 11.3. Si se detecta una diferencia en el cuarto carácter, una 'B', `fracaso[4]` apunta de vuelta a otra B, que por supuesto no coincide tampoco con el carácter actual del texto. Modifique el algoritmo 11.2 de modo que los valores de `fracaso` cumplan la condición planteada en la sección 11.3.3 (y que repetimos en el ejercicio anterior) *y también* la condición de que $p_r \neq p_k$. (Tenga cuidado; lo primero que a muchos se les ocurre no funciona.)

11.13 Reescriba los algoritmos KMP de modo que opere con entradas que son listas ligadas. Por sencillez, suponga que el tipo de los elementos de las listas es **`int`**. Utilice las operaciones del tipo de datos abstracto `ListaInt` de la sección 2.3.2 y suponga que T y P son objetos de esta clase.

11.14 ¿Cómo modificaría `explorKMP` (algoritmo 11.3) para que lea el texto de la entrada, carácter por carácter, en lugar de acceder a la cadena T? Suponga que la función `leer()` devuelve un **`int`**, que equivale al siguiente carácter de entrada, a menos que se haya llegado al fin del archivo, en cuyo caso devuelve -1. ¿Se necesitan todas las capacidades del TDA `Texto` propuesto en el ejercicio 11.3? Explique por qué sí o por qué no.

Sección 11.4 El algoritmo Boyer-Moore

11.15 Enumere los valores del arreglo `saltoCar` si se ejecuta el algoritmo Boyer-Moore con los patrones siguientes, suponiendo que el alfabeto es $\{A, B, \ldots, Z\}$.

a. *ABRACADABRA*

b. *ASTRACASTRA*

11.16 Enumere los valores de los arreglos `sufijo` y `saltoCotejo` si se ejecuta el algoritmo Boyer-Moore con los patrones siguientes.

a. *AAAB*

b. *AABAACAABABA*

c. *ABRACADABRA*

d. *ASTRACASTRA*

11.17 Como mostró el ejemplo 11.5, si sólo se usan los valores de `saltoCar`, sin emplear `saltoCotejo`, se puede efectuar una exploración muy rápida. Sin embargo, no podemos simplemente sustituir el enunciado

```
j += max(saltoCar[T[j]], saltoCotejo[k])
```

del algoritmo 11.6 por

```
j += saltoCar[T[j]].
```

¿Por qué no? ¿Qué otro cambio (pequeño) se necesita para que funcione el algoritmo de exploración?

11.18 Recuerde que el patrón $P =$ '$A \cdots AB$' ($m - 1$ A es seguidas de una B) y la cadena de texto $T =$ '$A \cdots A$' (n A es) son una entrada de peor caso para el algoritmo 11.1.

a. Dé los valores de los arreglos `saltoCar`, `sufijo` y `saltoCotejo` para P suponiendo que el alfabeto es $\{A, B, \ldots, Z\}$

b. ¿Cuántas comparaciones de caracteres efectúa exactamente `explorBM` para explorar T en busca de una ocurrencia de P?

11.19 Suponga que el texto se va leyendo conforme se necesita, un carácter a la vez. Sugiera una fórmula que relacione k, m y `saltoCotejo[k]` con el número de caracteres de texto nuevos que se necesitan cuando se detecta una diferencia en p_k.

11.20 Suponga que P y T son cadenas de bits.

a. Muestre los valores de los arreglos `saltoCar`, `sufijo` y `saltoCotejo` para el patrón 1101101011.

b. Con cadenas de bits en general, ¿cuál produce los "saltos" más largos, `saltoCar` o `saltoCotejo`?

Sección 11.5 Cotejo aproximado de cadenas

11.21 Un algoritmo para hallar una cadena que coincida exactamente sólo necesita decirnos en qué punto del texto principia el patrón, o dónde termina. Podemos determinar dónde termina la coincidencia en el texto, en caso de no especificarse, porque conocemos la longitud del patrón. No sucede así en el caso de un cotejo aproximado porque no sabemos cuántos caracteres del patrón o del texto faltan. Muestre cómo modificar o extender el algoritmo que detecta coincidencias k-aproximadas de modo que nos indique dónde principia la coincidencia aproximada del patrón en T.

11.22

a. Escriba el algoritmo completo para un cotejo k-aproximado. ¿Cuánto espacio requiere?

b. Muestre cómo usar una versión del TDA `ColaAbierta`, introducido en el ejercicio 11.4, para evitar que se ocupe una cantidad de espacio que dependa de n. *Sugerencia:* Haga que los elementos almacenados en la cola abierta sean arreglos de $m + 1$ enteros, correspondientes a las columnas de la tabla D. ¿Cuántas columnas como máximo necesitan estar disponibles en cualquier momento dado?

Problemas adicionales

11.23 Reescriba los tres algoritmos de exploración (algoritmos 11.1, 11.3 y 11.6) de modo que encuentren todas las ocurrencias del patrón en el texto.

11.24 P es una cadena de caracteres (de longitud m) que consiste en letras y cuando menos un asterisco ('$*$'). El asterisco es un carácter "comodín"; puede coincidir con cualquier sucesión de cero o más caracteres. Por ejemplo, si P = '*c*∗*años*' y T = '*felizcumpleaños*', habrá una coincidencia que principia en la '*c*' y termina en la '*s*'; el asterisco "coincide" con '*umple*'. Escriba un algoritmo que halle una coincidencia de P en una cadena de texto T (consistente en n caracteres), si la hay, y cite una cota superior para el orden de su tiempo de peor caso.

11.25 Sean $X = x_1x_2 \cdots x_n$ y $Y = y_1y_2 \cdots y_n$ dos cadenas de caracteres. Decimos que X es un *desplazamiento cíclico* de Y si existe algún r tal que $X = y_{r+1} \cdots y_n y_1 \cdots y_r$. Escriba un algoritmo $O(n)$ para determinar si X es un desplazamiento cíclico de Y.

11.26

a. Escriba un algoritmo eficiente para determinar si una cadena (larga) de texto contiene 25 espacios en blanco consecutivos. (No presente una copia exacta de un algoritmo del texto; adecúela.)

★ **b.** Construya un ejemplo de peor caso (o casi de peor caso) para su algoritmo. ¿Cuántas comparaciones de caracteres se efectúan en este caso?

c. Suponga que la cadena de texto contiene texto ordinario en español en el que las palabras y oraciones están separadas por espacios en blanco pero casi nunca hay dos espacios en blanco juntos. Si la longitud del texto es n, ¿cuántas comparaciones de caracteres efectuará aproximadamente su algoritmo?

★ **11.27** Investigue el problema de hallar cualquiera de un conjunto finito de patrones en una cadena de texto. ¿Puede extender cualquiera de los algoritmos de este capítulo para tener un algoritmo que haga algo mejor que buscar por separado cada uno de los patrones?

Programas

1. Implemente los tres algoritmos de búsqueda de cadenas exactas, incluyendo un contador del número de comparaciones de caracteres efectuadas; ejecute un conjunto grande de casos de prueba y compare los resultados. Utilice las técnicas de los ejercicios 11.3 y 11.4 para manejar el retroceso y el salto hacia adelante en el texto, para no tener que almacenar todo el texto en la memoria.
2. Escriba un programa para el algoritmo de cotejo k-aproximado, almacenando cuando más dos columnas a la vez. Incluya las mejoras del ejercicio 11.21.

Notas y referencias

Crochemore y Rytter (1994) es un libro sobre algoritmos de texto en general. Incluye los algoritmos Knuth-Morris-Pratt y Boyer-Moore y el cotejo k-aproximado. Las referencias principales para los algoritmos que presentamos aquí son Knuth, Morris y Pratt (1977) y Boyer y Moore (1977). La primera fase del algoritmo 11.5 se basa en Knuth, Morris y Pratt (1977), y la segunda, en una idea atribuida a K. Mehlhorn por Smit (1982). Guibas y Odlyzko (1977), Galil (1979) y

Apostolico y Giancarlo (1986) presentan diversas versiones lineales de peor caso del algoritmo Boyer-Moore. Véase también Aho y Corasick (1975). Boyer y Moore y Smit presentan comparaciones empíricas de los algoritmos descritos en este capítulo. La gráfica de la figura 11.10 es de Smit (1982).

La sección 11.5 se basa en Wagner y Fischer (1974). Hall y Dowling (1980) es una reseña de técnicas para cotejo aproximado de cadenas. Landau y Vishkin (1986) contiene un algoritmo $O(kn)$ para el cotejo k-aproximado de cadenas.

12

Polinomios y matrices

12.1 Introducción

Los problemas que examinamos en este capítulo son la evaluación de polinomios (con y sin procesamiento previo de los coeficientes), la multiplicación de polinomios (como ilustración de la transformada discreta de Fourier) y la multiplicación de matrices y vectores. Las operaciones que suelen utilizarse para tales tareas son multiplicación y suma. Antes, las computadoras tardaban mucho más en multiplicar que en sumar, y algunos de los algoritmos que presentamos "mejoran" los métodos directos o más conocidos reduciendo el número de multiplicaciones a expensas de algunas sumas adicionales. Por tanto, su valor depende de los costos relativos de las dos operaciones. Otros algoritmos que presentamos reducen el número de ambas operaciones (si la entrada es grande).

Varios algoritmos de este capítulo emplean el método de dividir y vencer: la evaluación de un polinomio con procesamiento previo de los coeficientes (sección 12.2.3), el algoritmo de multiplicación de matrices de Strassen (sección 12.3.4) y la transformada rápida de Fourier (sección 12.4).

En este capítulo presentamos muchos resultados de cota inferior sin demostrarlos. En las Notas y referencias al final del capítulo se presentan más comentarios y referencias de esos resultados.

12.2 Evaluación de funciones polinómicas

Consideremos el polinomio $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ con coeficientes reales y $n \geq 1$. Supóngase que se nos dan los coeficientes $a_0, a_1, \ldots, a_n$ y que el problema consiste en evaluar $p(x)$. En esta sección examinaremos algunos algoritmos y cotas inferiores para este problema.

El número de multiplicaciones y sumas efectuadas podría parecer una medida razonable del trabajo, pero algunos algoritmos podrían usar división y resta y hacer menos multiplicaciones y sumas. Por ello, sobre todo al tratar cotas inferiores, consideraremos el número total de multiplicaciones y divisiones y el número total de sumas y restas.

12.2.1 Algoritmos

Los dos tipos de algoritmos se denotarán con $*/$ y $\pm$, respectivamente.

La forma obvia de resolver el problema es calcular cada término y sumarlo al total de los que ya se han calculado. El algoritmo siguiente hace eso.

Algoritmo 12.1 Evaluación de polinomios — Término por término

Entradas: Los coeficientes del polinomio $p(x)$ en el arreglo a; $n \geq 0$, el grado de p; y x, el punto en el que se evaluará p.

Salidas: El valor de $p(x)$.

```
float poli(float[] a, int n, float x)
    float p, potenciax;
    int i;
    p = a[0]; potenciax = 1;
    for (i = 1; i <= n; i ++)
       potenciax = potenciax * x;
       p += a[i] * potenciax;
    return p;
```

El algoritmo 12.1 efectúa $2n$ multiplicaciones y n sumas.

Método de Horner

¿Existe una forma mejor?, ¿se puede calcular $ab + ac$, dados a, b y c, con menos de dos multiplicaciones? Claro que sí: factorizándolo en la forma $a(b + c)$. De forma similar, la clave del método de Horner para evaluar $p(x)$ no es más que una factorización dada de p:

$$p(x) = (\cdots ((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0.$$

El cálculo se efectúa en un ciclo corto con sólo n multiplicaciones y n sumas.

Algoritmo 12.2 Evaluación de polinomios — Método de Horner

Entradas: a, n y x como en el algoritmo 12.1.

Salidas: El valor de $p(x)$.

```
float poliHorner(float[] a, int n, float x)
    float p;
    int i;
    p = a[n];
    for (i = n - 1; i >= 0; i --)
       p = p * x + a[i];
    return p;
```

Así pues, con sólo factorizar p hemos reducido el número de multiplicaciones a la mitad sin aumentar el número de sumas. ¿Es posible reducir más el número de multiplicaciones?, ¿se puede reducir el número de sumas?

12.2.2 Cotas inferiores para la evaluación de polinomios

Así como usamos árboles de decisión en un modelo abstracto para establecer cotas inferiores para el ordenamiento (y otros problemas), necesitamos un modelo para los algoritmos que evalúan polinomios (y otros problemas de cálculo relacionados). Recordemos que los algoritmos representados por los árboles de decisión funcionaban con entradas de tamaño fijo y no tenían ciclos. Aquí usaremos un modelo llamado *programas en línea recta*. Los programas efectúan una sucesión de operaciones aritméticas; no hay ciclos ni ramificaciones. Los operandos podrían ser

elementos del conjunto de entradas del problema, I, elementos de algún conjunto de constantes, C, y resultados intermedios ya calculados. Las constantes podrían parecer innecesarias —no usamos ninguna en los dos algoritmos para evaluar polinomios que examinamos— pero permitir constantes simplifica los argumentos de cota inferior, y cualquier cota inferior deducida con un modelo que permite constantes será válida para un modelo más restringido que no lo haga.

Formalmente, un programa en línea recta es una sucesión finita de pasos de la forma

$$s_i = q \; op \; r$$

donde q y r son entradas, constantes o los resultados de pasos anteriores; es decir, q y r están en $I \cup C \cup \{s_j \mid j < i\}$ y op es un operador aritmético. El último paso deberá calcular $p(x)$. Para el problema de evaluar un polinomio $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, el conjunto de entrada es $I = \{x, a_0, a_1, \ldots, a_n\}$. Las entradas deben considerarse indeterminadas, es decir, símbolos abstractos sin hacer supuestos acerca de sus valores.

Ejemplo 12.1 Un programa en línea recta para el método de Horner con $n = 2$

$$\begin{aligned} s_1 &= a_2 * x \\ s_2 &= s_1 + a_1 \\ s_3 &= s_2 * x \\ s_4 &= s_3 + a_0 \end{aligned}$$ ■

Es obvio que el número de pasos de un programa en línea recta es una medida razonable del trabajo que realiza. Casi todos los teoremas cuentan los pasos $*$ / (multiplicación, división) y $\pm$ (suma, resta) por separado. Ilustraremos las técnicas de demostración mostrando que, si no se permiten divisiones, un programa en línea recta para evaluar un polinomio de grado n debe efectuar por lo menos n pasos $\pm$. Se puede demostrar con un argumento similar pero más complicado que, si no se permiten divisiones, se necesitan por lo menos n multiplicaciones. También se sabe que, si se permiten divisiones, se requieren por lo menos n pasos $*/$. Por tanto, el método de Horner utiliza el número óptimo de pasos $*/$ y, dado que la división tarda por lo menos lo mismo que la multiplicación, emplea la mezcla óptima de estos dos operadores.

Decimos que un paso $s_i = q \; op \; r$ *usa* una entrada α si y sólo si $q = \alpha$ o $r = \alpha$, o $q = s_j$ para alguna $j < i$ y s_j usa α, o $r = s_j$ para alguna $j < i$ y s_j usa α. Dicho de otro modo, si "expandiéramos" s_i sustituyendo los resultados de pasos anteriores hasta que sólo quedaran entradas y constantes, α aparecería en la expresión. En el ejemplo 12.1, s_3 usa a_2, a_1 y x.

Lema 12.1 Un programa en línea recta (que usa sólo $*$, $+$ y $-$) para calcular $a_0 + \cdots + a_n$ debe tener por lo menos n pasos $\pm$.

Demostración La demostración es por inducción con n. Para $n = 0$, observamos que cualquier programa tiene por lo menos cero pasos $\pm$. Para $n > 0$, suponemos que P es un programa para $a_0 + \cdots + a_n$. La idea en que se basa la demostración consiste en sustituir a_n por 0 para producir un programa que calcule $a_0 + \cdots + a_{n-1}$, y luego usar la hipótesis de inducción. Sea

$$s_i = q \; op \; r$$

el primer paso $\pm$ que usa a_n. (Debe existir tal paso; de lo contrario el resultado del cálculo sería un múltiplo de a_n.) Puesto que ningún paso $\pm$ anterior usó a_n, q o r deben ser a_n mismo o un múltiplo de a_n. Si sustituimos a_n por 0 tendremos uno de los casos siguientes:

1. $s_i = q \pm 0$
2. $s_i = 0 + r$
3. $s_i = 0 - r$

En los casos 1 y 2, eliminamos este paso del programa y sustituimos s_i por q o r, respectivamente, en todos los demás pasos en los que aparece s_i. En el caso 3, sustituimos este paso por

$$si = -1 * r.$$

En todos los casos hemos eliminado un paso $\pm$. Sustituimos a_n por 0 en todos los pasos en que aparece. Ahora tenemos un programa que calcula $a_0 + \cdots + a_{n-1}$. Por la hipótesis inductiva, este programa tiene por lo menos $n - 1$ pasos $\pm$. Por tanto, el programa original P tenía al menos n pasos $\pm$. □

Teorema 12.2 Un programa en línea recta que sólo usa $*$, $+$ y $-$ para evaluar

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0,$$

donde $a_0, \ldots, a_n$ y x son entradas arbitrarias, debe tener por lo menos n pasos $\pm$.

Demostración Sea P un programa para calcular $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$. Sustituimos cada referencia a x por "1". Esto no altera el número de pasos $\pm$. El programa resultante calcula $a_0 + \cdots + a_n$, así que debe tener por lo menos n pasos $\pm$. □

12.2.3 Procesamiento previo de coeficientes

El procesamiento previo (también llamado acondicionamiento) de algunos de los datos de un problema implica, informalmente, que parte de las entradas se conocen con antelación y es posible escribir un programa especializado. Supóngase que un problema tiene las entradas I e I' y que denotamos con A un algoritmo para resolver ese problema. Cuando hablamos de someter I a procesamiento previo, nos referimos a hallar un algoritmo A_I con la entrada I' que produce la misma salida que A con las entradas I e I'. Así pues, el problema de procesamiento previo tiene dos partes: el algoritmo A_I que depende de I y un algoritmo que, con I como entrada, produce el algoritmo A_I. En términos rigurosos, A y A_I resuelven problemas distintos y, como veremos, pueden tener diferente complejidad.

En algunas situaciones es preciso evaluar un polinomio con un número grande de argumentos distintos. Un ejemplo es la aproximación a una función con una serie de potencias. En tales casos, el procesamiento previo de los coeficientes podría reducir el número de pasos $*/$ que se requieren para cada evaluación.

Sea $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, donde $n = 2^k - 1$ para alguna $k \geq 1$. Por tanto, p tiene 2^k términos, algunos de los cuales podrían ser cero. El procedimiento para evaluar $p(x)$ que describimos aquí emplea un método de divide y vencerás para factorizar p. Suponemos que p es mónico (es decir, que $a_n = 1$). La extensión del algoritmo al caso general se deja como ejercicio.

$$
\begin{array}{r|l|l}
 & \overbrace{x^{2^{k-1}-1}+a_{2^k-2}x^{2^{k-1}-2}+\cdots+a_{2^{k-1}}}^{q(x)} & \\
x^{2^{k-1}}+b\,\Big) & x^{2^k-1}+a_{2^k-2}x^{2^k-2}+\cdots+a_{2^{k-1}}x^{2^{k-1}} \;\vdots\; +a_{2^{k-1}-1}x^{2^{k-1}-1} & +\cdots+a_1x+a_0 \\
 & x^{2^k-1}+a_{2^k-2}x^{2^k-2}+\cdots+a_{2^{k-1}}x^{2^{k-1}} \;\vdots\; +bx^{2^{k-1}-1} & +\cdots+ba_{2^{k-1}} \\
\hline
 & 0 \;\vdots\; \underbrace{(a_{2^{k-1}-1}-b)x^{2^{k-1}-1}+\cdots+(a_0-ba_{2^{k-1}})}_{r(x)} &
\end{array}
$$

Figura 12.1 $p(x)$ dividido entre $x^j + b$

Si $n = 1$, entonces $p(x) = x + a_0$ y se evalúa efectuando una suma. Supóngase $n > 1$ y escribamos p como sigue para algunas j y b:

$$p(x) = (x^j + b)q(x) + r(x),$$

donde q y r son polinomios mónicos de grado $2^{k-1} - 1$ (es decir, con la mitad de términos que p, contando los términos cero, si los hay). Entonces podremos evaluar $p(x)$ llevando a cabo estos pasos:

1. Evaluar $q(x)$ y $r(x)$.
2. Calcular x^j.
3. Multiplicar $(x^j + b)$ por $q(x)$ y sumarle $r(x)$.

Puesto que q y r satisfacen las mismas condiciones que p —es decir, son mónicos y su grado es $2^m - 1$ para alguna m— podemos usar recursivamente el mismo esquema para evaluarlos. ¿Qué j y b debemos escoger para asegurar que q y r tengan las propiedades deseadas? Obviamente, $j =$ grado(p) − grado(q) $= 2^k - 1 - (2^{k-1} - 1) = 2^{k-1}$. Cabe señalar que, al ser j una potencia de 2, x^j se puede calcular con relativa rapidez. El valor correcto de b se hace evidente cuando dividimos $p(x)$ entre $x^j + b$ para obtener $q(x)$, el cociente, y $r(x)$, el residuo. Véase la figura 12.1. Para que r sea mónico, $a_{2^{k-1}-1} - b$ deberá ser 1, así que $b = a_{2^{k-1}-1}$. Por tanto, el algoritmo de procesamiento previo factoriza p como sigue:

$$p(x) = \left(x^{2^{k-1}} + (a_{2^{k-1}-1} - 1)\right) q(x) + r(x)$$

y factoriza q y r recursivamente con el mismo procedimiento. La factorización es total cuando q y r tienen grado 1. El ejemplo que sigue ilustra todo el procedimiento.

Ejemplo 12.2

Sea $p(x) = x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x + 1$. Entonces $k = 3$, $j = 2^{k-1} = 4$, $b = a_{2^{k-1}-1} - 1 = a_3 - 1 = 2$ y $x^j + b = x^4 + 2$. La figura 12.2(a) muestra el cálculo de $q(x)$ y $r(x)$. Entonces

$$p(x) = (x^4 + 2)(x^3 + 6x^2 + 5x + 4) + (x^3 - 10x^2 - 9x - 7).$$

$$
\begin{array}{r}
\overbrace{x^3+6x^2+5x+4}^{q(x)} \\
x^4+2\,\big)\;x^7+6x^6+5x^5+4x^4 \;\big|\; +3x^3+\;2x^2+\quad x+1 \\
\underline{\;x^7+6x^6+5x^5+4x^4\;} \;\big|\; \underline{+2x^3+12x^2+10x+8} \\
\qquad\qquad\qquad\qquad\qquad\quad \underbrace{x^3-10x^2-\;9x-7}_{r(x)}
\end{array}
$$

(a) Cálculo de $q(x)$ y $r(x)$.

$q(x) = x^3+6x^2+5x+4$
$k=2; \quad j=2^{k-1}=2$
$b = a_{2^{k-1}-1} - 1 = a_1 - 1 = 4$

$$
\begin{array}{r}
x\;+6 \\
x^2+4\,\big)\;x^3+6x^2 \;\big|\; +5x+\;4 \\
\underline{\;x^3+6x^2\;} \;\big|\; \underline{+4x+24} \\
\qquad\qquad\quad x-20
\end{array}
$$

(b) Factorización recursiva de $q(x)$: Así, $q(x) = (x^2 + 4)(x + 6) + (x - 20)$.

$r(x) = x^3-10x^2-9x-7$
$k=2; \quad j=2^{k-1}=2$
$b = a_{2^{k-1}-1} - 1 = a_1 - 1 = -10$

$$
\begin{array}{r}
x\;-10 \\
x^2-10\,\big)\;x^3-10x^2 \;\big|\; -\;9x-\;7 \\
\underline{\;x^3-10x^2\;} \;\big|\; \underline{-10x+100} \\
\qquad\qquad\quad x-107
\end{array}
$$

(c) Factorización recursiva de $r(x)$: Así, $r(x) = (x^2 - 10)(x - 10) + (x - 107)$.

Figura 12.2 Detalles de cálculo del ejemplo 12.2

Ahora factorizamos $q(x)$ y $r(x)$ de la misma manera, como se muestra en las figuras 12.2(b) y 12.2(c). Entonces,

$$p(x) = (x^4 + 2)\Big((x^2 + 4)(x + 6) + (x - 20)\Big) + \Big((x^2 - 10)(x - 10) + (x - 107)\Big).$$

Utilizando esta fórmula, la evaluación de $p(x)$ requiere cinco multiplicaciones: tres que aparecen explícitamente en la factorización y dos para calcular x^2 y x^4. El método de Horner habría requerido siete. Observemos, empero, que se efectúan 10 sumas (y restas) en lugar de siete. ■

Análisis de la evaluación de polinomios con procesamiento previo de coeficientes

Es fácil contar el número de operaciones que se efectúan para evaluar $p(x)$ (después de realizado el procesamiento previo) considerando los tres pasos que se usan para describir el procedimiento:

1. Evaluar $q(x)$ y $r(x)$ recursivamente.
   Esto sugiere el uso de una ecuación de recurrencia.

2. Calcular x^j.
   La j más grande empleada es 2^{k-1} y podemos calcular $x^2, x^4, x^8, \ldots, x^{2^{k-1}}$ efectuando $k - 1$ multiplicaciones.
3. Multiplicar $(x^j + b)$ por $q(x)$ y sumarle $r(x)$.
   Una multiplicación y dos sumas.

Sea $M(k)$ el número de multiplicaciones que se efectúan para evaluar un polinomio mónico de grado $2^k - 1$, *sin* contar las potencias de x (ya que pueden calcularse una vez y usarse cuando se necesiten). Sea $S(k)$ el número de sumas (y restas). Entonces:

$$M(1) = 0$$
$$M(k) = 2M(k-1) + 1 \qquad \text{para } k > 1$$

y

$$S(1) = 1$$
$$S(k) = 2S(k-1) + 2 \qquad \text{para } k > 1.$$

Si expandimos $M(k)$ unas cuantas veces, vemos que

$$M(k) = 4M(k-2) + 2 + 1 = 8M(k-3) + 4 + 2 + 1 = \sum_{i=0}^{k-2} 2^i = 2^{k-1} - 1.$$

El número total de multiplicaciones, pues, es $2^{k-1} - 1 + k - 1$. (El término $k - 1$ es para calcular potencias de x.) Puesto que $n = 2^k - 1$, el número de multiplicaciones es $n/2 + \lg(n+1) - 3/2$, o aproximadamente $n/2 + \lg n$. Es fácil demostrar que $S(k) = (3n-1)/2$. (Recomendamos al lector verificar que estas fórmulas describen el número de operaciones efectuadas en el ejemplo.)

Sea que se ahorre tiempo o no eliminando $n/2 - \lg n$ multiplicaciones efectuando $n/2$ sumas extra, hemos ilustrado un punto importante: las cotas inferiores que se han obtenido para un problema sin procesamiento previo, en este caso n pasos $*/$ para evaluar un polinomio de grado n, podrían dejar de tener validez. Las operaciones específicas que se permitan en el procesamiento previo (por ejemplo, división de polinomios, como en este caso, o la obtención de raíces de polinomios) también pueden afectar el número de operaciones requeridas. Se ha establecido una cota inferior de $\lceil n/2 \rceil$ pasos $*/$ para la evaluación de polinomios, permitiendo diversas operaciones de procesamiento previo.

12.3 Multiplicación de vectores y matrices

Iniciamos con un repaso de los métodos muy conocidos para multiplicar matrices y vectores, observando el número de operaciones que efectúan esos métodos y dando las cotas inferiores conocidas para el número de multiplicaciones y divisiones (pasos $*/$). Luego veremos algunas estrategias más avanzadas. Una de ellas, que adopta un enfoque de divide y vencerás, puede mejorar el orden asintótico del procedimiento directo para multiplicar matrices.

En toda esta sección usaremos letras mayúsculas para los nombres de vectores y matrices, y las letras minúsculas correspondientes para sus componentes, que son números reales.

12.3.1 Repaso de algoritmos estándar

Sean $V = (v_1, v_2, \ldots v_n)$ y $W = (w_1, w_2, \ldots, w_n)$ dos n-vectores, es decir, vectores con n componentes cada uno. El producto punto de V y W, denotado por $V \cdot W$, se define como $V \cdot W = \sum_{i=1}^{n} v_i w_i$. El cálculo de $V \cdot W$ que la definición implica requiere n multiplicaciones y $n - 1$ sumas. Se ha demostrado que, incluso si uno de los vectores se conoce con antelación y se permite cierto procesamiento previo de sus componentes, se requieren por lo menos n pasos $*/$ en el peor caso. Por tanto, el cálculo directo de los productos punto es óptimo.

Sea A una matriz $m \times n$ y sea V un n-vector. Sea W el producto AV. Por definición, el i-ésimo componente de W es el producto punto de la i-ésima fila de A por V. Es decir, para $1 \leq i \leq m$, $w_i = \sum_{j=1}^{n} a_{ij} v_j$. El cálculo de AV que la definición implica requiere mn multiplicaciones. Se sabe que esto es óptimo. El número de sumas efectuadas es $m(n - 1)$.

Sea A una matriz $m \times n$, sea B una matriz $n \times q$ y sea C el producto de A y B. Por definición, c_{ij} es el producto punto de la i-ésima fila de A por la j-ésima columna de B. Es decir, para $1 \leq i \leq m$ y $1 \leq j \leq q$, $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$. Si los elementos de C se calculan con el algoritmo acostumbrado para multiplicar matrices, es decir, como indica esta fórmula, se efectúan mnq multiplicaciones y $m(n - 1)q$ sumas. Para el asombro de quienes estudian el problema, los intentos por demostrar que se requieren mnq pasos $*/$ para multiplicar matrices fueron infructuosos, y finalmente se buscaron y hallaron algoritmos que efectúan menos pasos $*/$. Aquí presentaremos dos de ellos.

12.3.2 Multiplicación de matrices de Winograd

Supóngase que el producto punto de $V = (v_1, v_2, v_3, v_4)$ y $W = (w_1, w_2, w_3, w_4)$ se calcula con la fórmula siguiente:

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4.$$

Obsérvese que en las últimas cuatro multiplicaciones sólo intervienen componentes de V o sólo componentes de W. Sólo hay dos multiplicaciones en las que intervienen componentes de ambos vectores. Obsérvese también que la fórmula depende de la conmutatividad de la multiplicación; por ejemplo, aprovecha el hecho de que $w_2 v_2 = v_2 w_2$. Por tanto, la fórmula no sería válida si la multiplicación de los componentes no fuera conmutativa; en particular, no sería válida si los componentes fueran matrices.

Generalizando a partir del ejemplo, cuando n es par (digamos, $n = 2p$),

$$V \cdot W = \sum_{i=1}^{p} (v_{2i-1} + w_{2i})(v_{2i} + w_{2i-1}) - \sum_{i=1}^{p} v_{2i-1} v_{2i} - \sum_{i=1}^{p} w_{2i-1} w_{2i}. \tag{12.1}$$

Si n es impar, hacemos $p = \lfloor n/2 \rfloor$ y sumamos el término final $v_n w_n$ a la ecuación (12.1). En cada sumatoria se efectúan p, o $\lfloor n/2 \rfloor$ multiplicaciones, así que en total se efectúan $3\lfloor n/2 \rfloor$ multiplicaciones. Esto es peor que el método directo para calcular el producto punto. Incluso si uno de los vectores se conoce con antelación y la segunda y tercera sumatorias se pueden considerar como procesamiento previo, de todos modos se efectuarían n multiplicaciones. Si *ambos* vectores se conocen con antelación, todo el cálculo podría considerarse como procesamiento previo, ¡con lo que se elimina todo el problema! Entonces, ¿qué hemos ganado al examinar una fórmula más complicada para obtener el producto punto?

Supóngase que debemos multiplicar la matriz A de $m \times n$ y la matriz B de $n \times q$. Cada fila de A interviene en q productos punto, uno con cada columna de B, y cada columna de B interviene en m productos punto, uno con cada fila de A. Por tanto, términos como las últimas dos sumatorias de la ecuación (12.1) se pueden calcular una vez para cada fila de A y cada columna de B y usarse muchas veces.

Algoritmo 12.3 Multiplicación de matrices de Winograd

Entradas: A, B, m, n y q, donde A y B son matrices de $m \times n$ y $n \times q$, respectivamente.

Salidas: La matriz $C = AB$. La matriz C de $m \times q$ se pasa como parámetro y el algoritmo la llena.

```
void winograd(float[][] A, float[][] B, int n, int m, int q, float[][] C)
    int i, j, k;
    int p = n/2;
    float[] termFila = new float[m+1];
    float[] termCol = new float[q+1];
    // Estos arreglos son para los resultados del "preprocesamiento"
    // de las filas de A y las columnas de B.

    // "Preprocesar" filas de A.
    for (i = 1; i <= m; i++)
        termFila[i] = ∑_{j=1}^{p} a_{i,2j-1} * a_{i,2j};

    // "Preprocesar" columnas de B.
    for (i = 1; i <= q; i++)
        termCol[i] = ∑_{j=1}^{p} b_{2j-1,i} * b_{2j,i};

    // Calcular elementos de C.
    for (i = 1; i <= m; i++)
        for (j = 1; j <= q; j++)
            c_{ij} = ∑_{k=1}^{p} (a_{i,2k-1} + b_{2k,j}) * (a_{i,2k} + b_{2k-1,j})
                - termFila[i] - termCol[i];

    // Si n es impar, sumar un último término a cada elemento de C.
    if (odd(n))
        for (i = 1; i <= m; i++)
            for (j = 1; j <= q; j++)
                c_{ij} = c_{ij} + a_{in} * b_{nj};
```

Análisis

Supóngase que n es par. (El caso de n impar se deja como ejercicio.) Contamos primero las multiplicaciones. El procesamiento de las filas de A efectúa mp, el de las columnas de B efectúa qp, y el cálculo de los elementos de C efectúa mqp multiplicaciones. El total, dado que $p = n/2$, es $(mnq/2) + (n/2)(q + m)$. Si A y B son matrices cuadradas, ambas de $n \times n$, el algoritmo de Wi-

nograd efectuará $(n^3/2) + n^2$ multiplicaciones en lugar de las n^3 acostumbradas. (Véase el algoritmo 1.2.) La diferencia es significativa incluso cuando n es pequeño. Lo malo es que el algoritmo de Winograd efectúa pasos $\pm$ extra. Contaremos los pasos $\pm$ como sigue:

Procesamiento de filas de A:	$m(p-1)$
Procesamiento de columnas de B:	$q(p-1)$
Cálculo de elementos de C:	Para cada uno de los mq elementos de C hacemos:

dos sumas en cada término de la sumatoria:	$2p$
sumar los términos de la sumatoria:	$p-1$
restar `termFila[i]` y `termCol[j]`:	2

Por tanto, para calcular los elementos de C el algoritmo efectúa $mq(3p+1)$ pasos $\pm$, y el total, suponiendo que n es par, es de $(3/2)(mnq) + (n/2)(m+q) + mq - m - q$. En el caso de matrices cuadradas de $n \times n$, en el que es un poco más fácil comparar los algoritmos, el algoritmo de Winograd efectúa $(3/2)n^3 + 2n^2 - 2n$ pasos $\pm$ en lugar de los $n^3 - n^2$ acostumbrados.

Obsérvese que el algoritmo de Winograd contiene menos instrucciones que requieren incrementar y probar contadores de ciclos, que el método acostumbrado. Por otra parte, el algoritmo de Winograd maneja subíndices más complejos y pide obtener elementos de matrices con mayor frecuencia. Estas diferencias se explorarán en los ejercicios.

12.3.3 Cotas inferiores para la multiplicación de matrices

El algoritmo de Winograd muestra que podemos multiplicar matrices de $m \times n$ y $n \times q$ efectuando menos de mnq multiplicaciones. ¿Cuántos pasos $*/$ se necesitan? ¿Esa cifra está en $\Theta(mnq)$ o es posible eliminar el término cúbico? La mejor cota inferior conocida es sorprendentemente baja: mn, o n^2 si las matrices son cuadradas. Ya dijimos antes que se necesitan mn pasos $*/$ para multiplicar una matriz de $m \times n$ por un n-vector. Cabría esperar que la multiplicación de matrices sea por lo menos igual de difícil, y por ende que requiera al menos el mismo número de pasos $*/$; la figura 12.3 ilustra que ello es cierto mostrando que es posible usar un algoritmo de multiplicación de matrices para obtener el producto de una matriz por un vector. (Desde luego, los dos problemas son uno solo si $q = 1$.) Ningún algoritmo conocido para multiplicar matrices efectúa sólo mn

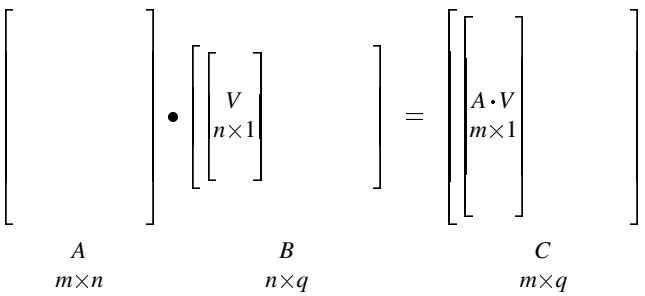


Figura 12.3 Cota inferior para la multiplicación de matrices

pasos $*/$. No obstante, existen algoritmos que, si las matrices son grandes, realizan un número significativamente menor de multiplicaciones *y* de pasos $\pm$ que el de Winograd.

12.3.4 Multiplicación de matrices de Strassen

En el resto de la sección supondremos que las matrices a multiplicar son cuadradas (de $n \times n$) y las llamaremos A y B. El algoritmo de Strassen es de divide y vencerás. La clave del algoritmo es un método para multiplicar matrices 2×2 efectuando siete multiplicaciones en lugar de las ocho acostumbradas. (El algoritmo de Winograd también efectúa ocho.) Para $n = 2$, primero calculamos las siete cantidades siguientes, cada una de las cuales requiere exactamente una multiplicación:

$$\begin{aligned} x_1 &= (a_{11} + a_{22}) * (b_{11} + b_{22}) & x_5 &= (a_{11} + a_{12}) * b_{22} \\ x_2 &= (a_{21} + a_{22}) * b_{11} & x_6 &= (a_{21} - a_{11}) * (b_{11} + b_{12}) \\ x_3 &= a_{11} * (b_{12} - b_{22}) & x_7 &= (a_{12} - a_{22}) * (b_{21} + b_{22}) \\ x_4 &= a_{22} * (b_{21} - b_{11}) \end{aligned} \tag{12.2}$$

Sea $C = AB$. Los elementos de C son

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} & c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + b_{22}b_{21} & c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

y se calculan como sigue:

$$\begin{aligned} c_{11} &= x_1 + x_4 - x_5 + x_7 & c_{12} &= x_3 + x_5 \\ c_{21} &= x_2 + x_4 & c_{22} &= x_1 + x_3 - x_2 + x_6 \end{aligned} \tag{12.3}$$

Así, podemos multiplicar matrices 2×2 haciendo siete multiplicaciones y 18 sumas. Es crucial para el algoritmo de Strassen que no se use la conmutatividad de la multiplicación en las fórmulas de las ecuaciones (12.2), para poder aplicarlas a matrices cuyos componentes también sean matrices. Sea n una potencia de 2. El método de Strassen consiste en dividir A y B en cuatro matrices de $n/2 \times n/2$ cada una, como se muestra en la figura 12.4, y multiplicarlas empleando las fórmulas de las ecuaciones (12.2) y (12.3); las fórmulas se usan recursivamente para multiplicar las matrices componentes. Antes de considerar extensiones al caso en que n no es una potencia de 2, calcularemos el número de multiplicaciones y pasos $\pm$ efectuados.

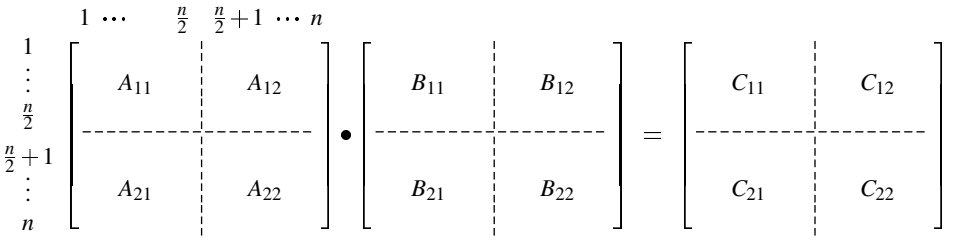


Figura 12.4 Partición para la multiplicación de matrices de Strassen

Supóngase que $n = 2^k$ para alguna $k \geq 0$. Sea $M(k)$ el número de multiplicaciones (de los componentes matriciales subyacentes, es decir, números reales) que el método de Strassen efectúa con matrices de $n \times n$. Entonces, dado que las fórmulas de la ecuación (12.2) efectúan siete multiplicaciones de matrices de $2^{k-1} \times 2^{k-1}$,

$$\begin{aligned} M(0) &= 1 \\ M(k) &= 7M(k-1) \qquad \text{para } k > 0. \end{aligned}$$

Esta ecuación de recurrencia es muy fácil de resolver. $M(k) = 7^k$, y $7^k = 7^{\lg n} = n^{\lg 7} \approx n^{2.81}$. Por tanto, el número de multiplicaciones está en $o(n^3)$.

Sea $P(k)$ el número de pasos $\pm$ efectuados. Es obvio que $P(0) = 0$. Hay 18 pasos $\pm$ en las fórmulas de las ecuaciones (12.2) y (12.3), así que $P(1) = 18$. Para $k \geq 1$, multiplicar matrices de $2^k \times 2^k$ implica 18 sumas de matrices de $2^{k-1} \times 2^{k-1}$, más todos los pasos $\pm$ efectuados por las siete multiplicaciones de matrices de las ecuaciones (12.2). Entonces,

$$\begin{aligned} P(0) &= 0 \\ P(k) &= 18(2k-1)2 + 7P(k-1) \qquad \text{para } k > 0. \end{aligned}$$

Podemos expandir el árbol de recursión (véase la sección 3.7.2) para ver qué aspecto tienen las sumas de filas, o expandir la ecuación de recurrencia para ver qué aspecto tienen los términos.

$$\begin{aligned} P(k) &= 18(2^{k-1})^2 + 7P(k-1) \\ &= 18(2^{k-1})^2 + 7 \cdot 18(2^{k-2})^2 + 7^2 P(k-2) \\ &= 18(2^{k-1})^2 + 7 \cdot 18(2^{k-2})^2 + 7^2 \cdot 18(2^{k-3})^2 + 7^3 P(k-3) \\ &\;\;\vdots \\ &= 18 \cdot 2^{2(k-1)} + 7 \cdot 18 \cdot 2^{2(k-2)} + 7^2 \cdot 18 \cdot 2^{2(k-3)} + \cdots + 7^{k-1} \cdot 18 \end{aligned}$$

Se ha desarrollado una serie geométrica con razón $r = 7/4$, así que la sumatoria está en Θ de su término más grande (véase el teorema 1.13). El término más grande es $18 \cdot 7^{k-1}$, que está en $\Theta(7^k)$. Como acabamos de ver en el caso de $M(k)$, esto está en $\Theta(n^{\lg 7})$.

Si queremos un valor más exacto, podemos usar la ecuación (1.9).

$$\begin{aligned} P(k) &= \sum_{i=0}^{k-1} 7^i \, 18(2^{k-i-1})^2 = 18(2^k)^2 \sum_{i=0}^{k-1} \frac{7^i}{(2^{i+1})^2} \\ &= \frac{9}{2} 2^{2k} \sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i = \frac{9}{2} 2^{2k} \left(\frac{\left(\frac{7}{4}\right)^k - 1}{\frac{7}{4} - 1} \right) = 6 \cdot 7^k - 6 \cdot 4^k \\ &\approx 6n^{2.81} - 6n^2. \end{aligned}$$

Así pues, si n es grande, este algoritmo efectúa aproximadamente $7\, n^{2.81}$ operaciones aritméticas.

Si n no es una potencia de 2, es preciso usar alguna extensión del algoritmo de Strassen, lo que implica más trabajo. Hay dos enfoques sencillos, pero ambos pueden ser muy lentos. La pri-

	Algoritmo acostumbrado	Algoritmo de Winograd	Algoritmo de Strassen (sin mejoras)
Multiplicaciones	n^3	$\frac{1}{2}n^3 + n^2$	$7^k \approx n^{2.81}$, donde $n = 2^k$
Sumas/restas	$n^3 - n^2$	$\frac{3}{2}n^3 + 2n^2 - 2n$	$6 \cdot 7^k - 6 \cdot 4k \approx 6n^{2.81} - 6n^2$, donde $n = 2^k$
Total	$2n^3 - n^2$	$2n^3 + 3n^2 - 2n$	$7n^{\lg 7} - 6n^2 \approx 7n^{2.81} - 6n^2$

Tabla 12.1 Comparación de métodos para multiplicar matrices de $n \times n$

mera posibilidad es añadir filas y columnas de ceros extra para que la dimensión sea una potencia de 2. La segunda es usar las fórmulas de Strassen si la dimensión de las matrices es par y el algoritmo acostumbrado si es impar. Otra posibilidad más complicada es modificar el algoritmo de modo que en cada nivel de la recursión, si las matrices a multiplicar tienen dimensiones impares, se añadan una fila y una columna extra. Strassen describió una cuarta estrategia que combina las ventajas de las primeras dos y también mejora el desempeño cuando n es una potencia de 2. Las matrices se incrustan en otras (posiblemente) más grandes de dimensión $2^k m$, donde $k = \lfloor \lg n - 4 \rfloor$ y $m = \lceil n/2^k \rceil$. Las fórmulas de Strassen se usan recursivamente hasta que las matrices a multiplicar son de $m \times m$, y luego se aplica el método acostumbrado. Con esta mejora, el número total de operaciones aritméticas efectuadas con elementos de las matrices es menor que $4.7n^{\lg 7}$ (véase el ejercicio 12.11).

En la tabla 12.1 se comparan las cantidades de operaciones aritméticas que efectúan los tres métodos para multiplicar matrices de $n \times n$. Si n es grande, el algoritmo de Strassen efectúa menos multiplicaciones *y* menos pasos $\pm$ que cualquiera de los otros métodos. En la práctica, empero, debido a su naturaleza recursiva, la implementación de este algoritmo requiere muchas tareas de "contabilidad" que podrían ser lentas y son complicadas. Los otros algoritmos, más sencillos, son más eficientes si n es de tamaño moderado.

La importancia primordial del algoritmo de Strassen es que rompió la barrera de $\Theta(n^3)$ para la multiplicación de matrices *y* la barrera de $\Theta(n^3)$ para varios otros problemas de matrices. Todos esos problemas, que incluyen la inversión de matrices, el cálculo de determinantes y la resolución de sistemas de ecuaciones lineales simultáneas, tienen soluciones $\Theta(n^3)$ muy conocidas, así que ellos también pueden resolverse en tiempo $O(n^{\lg 7})$. El resultado de Strassen se ha mejorado teóricamente varias veces. Existe un algoritmo para multiplicar matrices con tiempo de ejecución en $O(n^{2.376})$. La cota inferior de n^2 multiplicaciones no se ha aumentado; todavía no se sabe si es posible multiplicar matrices con $\Theta(n^2)$ pasos.

⋆ 12.4 La transformada rápida de Fourier y convolución

La transformada de Fourier tiene amplias aplicaciones en ingeniería, física y matemáticas. Su versión discreta se usa en problemas de interpolación, en la obtención de soluciones para ecuaciones diferenciales parciales, en el diseño de circuitos, en cristalografía y, muy extensamente, en el procesamiento de señales. Éste fue uno de los primeros problemas con los que se usó la estrategia de

divide y vencerás para desarrollar un algoritmo con orden asintótico más bajo que el cálculo directo. El algoritmo mejorado se denomina transformada rápida de Fourier.

Este algoritmo tuvo un impacto considerable porque muchos otros cálculos matemáticos se pueden expresar en términos de transformadas de Fourier. En algunos casos resulta más fácil convertir el problema natural en un problema de transformadas de Fourier, y efectuar dos transformadas de Fourier, que calcular una solución del problema original directamente. La convolución es un ejemplo.

Definición 12.1 Convolución

Sean U y V n-vectores con componentes indizados de 0 a $n - 1$. La *convolución* de U y V, denotada por $U \star V$ es, por definición, un n-vector W con componentes $w_i = \sum_{j=0}^{n-1} u_j v_{i-j}$, donde $0 \leq i \leq n - 1$ y los índices del miembro derecho se toman módulo n. ■

Por ejemplo, para $n = 5$,

$$\begin{aligned} w_0 &= u_0v_0 + v_1v_4 + u_2v_3 + u_3v_2 + u_4v_1 \\ w_1 &= u_0v_1 + u_1v_0 + u_2v_4 + u_3v_3 + u_4v_2 \\ &\vdots \\ w_4 &= u_0v_4 + u_1v_3 + u_2v_2 + u_3v_1 + u_4v_0. \end{aligned}$$

El problema de calcular la convolución de dos vectores surge de manera natural y frecuente en problemas de probabilidad, ingeniería y otras áreas. La multiplicación simbólica de polinomios, que se examina en esta sección, es un cálculo de convolución.

La transformada discreta de Fourier de un n-vector y la convolución de dos n-vectores se pueden calcular de manera directa realizando n^2 multiplicaciones y menos de n^2 sumas. Presentaremos un algoritmo de divide y vencerás para calcular la transformada discreta de Fourier ejecutando $\Theta(n \log n)$ operaciones aritméticas. Este algoritmo (que aparece en la literatura en muchas variaciones) se conoce como transformada rápida de Fourier, o FFT (por sus siglas en inglés). Luego usamos la FFT para calcular convoluciones en tiempo $\Theta(n \log n)$. Este ahorro de tiempo es muy valioso en las aplicaciones.

En toda esta sección todos los índices de matrices, arreglos y vectores principian en 0. Las *raíces de unidad* (también llamadas *raíces de* 1) complejas y algunas de sus propiedades elementales se usan en la FFT. Las definiciones básicas y las propiedades requeridas se repasan en el apéndice de esta sección (sección 12.4.3). Recomendamos a los lectores que no están familiarizados con los números complejos o las raíces n-ésimas de unidad leer el apéndice antes de continuar.

12.4.1 La transformada rápida de Fourier

La transformada discreta de Fourier transforma un n-vector complejo (es decir, un n-vector con componentes complejos) en otro n-vector complejo. Para transformar un n-vector real, basta con tratarlo como un n-vector complejo en el que todas las partes imaginarias son cero.

Definición 12.2 Transformada discreta de Fourier y la matriz F_n

Para $n \geq 1$, sea ω una raíz n-ésima primitiva de 1, y sea F_n la matriz de $n \times n$ con elementos $f_{ij} = \omega^{ij}$, donde $0 \leq i, j \leq n - 1$. La *transformada discreta de Fourier* del n-vector $P = (p_0, p_1, \ldots, p_{n-1})$ es el producto F_nP. ■

Los componentes de F_nP son

$$\begin{aligned}
&w^0p_0 + w^0p_1 + \cdots + w^0p_{n-2} + w^0p_{n-1}\\
&w^0p_0 + wp_1 + \cdots + w^{n-2}\,p_{n-2} + w^{n-1}\,p_{n-1}\\
&\quad\vdots\\
&w^0p_0 + w^ip_1 + \cdots + w^{i(n-2)}\,p_{n-2} + w^{i(n-1)}\,p_{n-1}\\
&\quad\vdots\\
&w^0p_0 + w^{n-1}p_1 + \cdots + w^{(n-1)(n-2)}\,p_{n-2} + w^{(n-1)(n-1)}p_{n-1}.
\end{aligned}$$

Reescrito en una forma un poco distinta, el i-ésimo componente es

$$p_{n-1}(w^i)^{n-1} + p_{n-2}(w^i)^{n-2} + \cdots + p_1w^i + p_0.$$

Así pues, si interpretamos los componentes de P como coeficientes del polinomio $p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \cdots + p_1x + p_0$, entonces el i-ésimo componente es $p(\omega^i)$ y el cálculo de la transformada discreta de Fourier implica evaluar el polinomio $p(x)$ en ω^0, ω, ω^2, …, ω^{n-1}, es decir, en cada una de las raíces n-ésimas de 1. Enfocaremos el problema desde este punto de vista. Primero desarrollaremos un algoritmo recursivo de divide y vencerás y luego lo examinaremos detenidamente para eliminar la recursión. Supondremos que $n = 2^k$ para alguna $k \geq 0$. (Se puede ajustar el algoritmo si se va a usar con una n que no es potencia de 2.)

La estrategia de divide y vencerás consiste en dividir el problema en ejemplares más pequeños, resolverlos, y utilizar las soluciones para obtener la solución del ejemplar actual. Aquí, para evaluar p en n puntos, evaluamos dos polinomios más pequeños en un subconjunto de los puntos y luego combinamos los resultados de forma apropiada. Recordemos que $\omega^{n/2} = -1$ y por tanto, para $0 \leq j \leq (n/2) - 1$, $\omega^{(n/2)+j} = -\omega^j$. Agrupamos los términos de $p(x)$ que tienen potencias pares y los que tienen potencias impares así:

$$p(x) = \sum_{i=0}^{n-1} p_ix^i = \sum_{i=0}^{n/2-1} p_{2i}x^{2i} + x\sum_{i=0}^{n/2-1} p_{2i+1}x^{2i}.$$

Definimos

$$p_{pares}(x) = \sum_{i=0}^{n/2-1} p_{2i}x^i \quad \text{y} \quad p_{impares}(x) = \sum_{i=0}^{n/2-1} p_{2i+1}x^i.$$

Entonces

$$p(x) = p_{pares}(x^2) = xp_{impares}(x2) \quad \text{y} \quad p(-x) = p_{pares}(x^2) - xp_{impares}(x^2). \tag{12.4}$$

La ecuación (12.4) muestra que para evaluar p en $1, \omega, \ldots, \omega^{(n/2)-1}, -1, -\omega, \ldots, -\omega^{(n/2)-1}$, basta con evaluar p_{pares} y $p_{impares}$ en $1, \omega^2, \ldots, (\omega^{(n/2)-1})^2$ y luego efectuar $n/2$ multiplicaciones (para $xp_{impares}(x^2)$) y n sumas y restas. Los polinomios p_{pares} y $p_{impares}$ se pueden evaluar recursivamente con el mismo esquema. Es decir, son polinomios de grado $n/2 - 1$ y se les evaluará en las raíces $n/2$-ésimas de unidad: $1, \omega^2, \ldots, (\omega^{(n/2)-1})^2$. Es evidente que, si el polinomio a evaluar es una constante, no se realizará trabajo.

Después se continúa con el algoritmo recursivo.

Algoritmo 12.4 Transformada rápida de Fourier (versión recursiva)

Entradas: El vector $P = (p_0, p_1, \ldots, p_{n-1})$, como arreglo **`Complex`** con n elementos, donde $n = 2^k$; un entero $k > 0$; y un entero $m > 0$. (Para procesar un vector de **`float`**, se le copia en la parte real del arreglo **`Complex`** P y se ponen en 0 todas las partes imaginarias.)

Salidas: La transformada discreta de Fourier de P almacenada en el arreglo **`Complex`** `transformada`. Este arreglo (con índices $0, \ldots, 2^k - 1$) se pasa como parámetro y el algoritmo lo llena.

Comentario: Suponemos que la clase **`Complex`** ofrece operaciones aritméticas con complejos, a fin de simplificar el pseudocódigo. Esta clase no existe actualmente en Java.

Suponemos que las raíces 2^k-ésimas de 1: $\omega^0, \omega, \ldots, \omega^{2^k-1}$, están almacenadas en el arreglo global `omega` en el orden en que se dan aquí. Usamos m para seleccionar raíces de este arreglo. El procedimiento `FFTrecursiva` se invocaría inicialmente con $m = 1$. En general, el conjunto que consiste en cada m-ésimo elemento, es decir, $\omega^0, \omega^m, \omega^{2m}, \ldots,$ es el conjunto de raíces $(2^k/m)$-ésimas de 1.

```
void FFTrecursiva(Complex[] P, int k, int m, Complex[] transformada)
   if (k==0)
     transformada[0] = P[0];

   else
     int n = 2^k;
     Complex[] pares = new Complex[n/2];
     Complex[] pares = new Complex[n/2];
     Complex xPImpares;
     int j;
     // Evaluar p_pares en las raíces 2^(k-1)-ésimas de 1.
     FFTrecursiva((p_0, p_2, ..., p_(2^k-2)), k-1, 2m, pares);
     // Evaluar p_impares en las raíces 2^(k-1)-ésimas de 1.
     FFTrecursiva((p_0, p_3, ..., p_(2^k-1)), k-1, 2m, impares);
     for (j = 0; j <= 2^(k-1) - 1; j ++)
        // Evaluar p(ω^j) y p(w^(2^(k-1) + j)).
        xPImpares = omega[m*j] * impares[j];
        // Calcular p(ω^j).
        transformada[j] = impares[j] + xPImpares;
        // Calcular p(w^(2^(k-1) + j)).
        transformada[2^(k-1) + j] = impares[j] - xPImpares;
```

La naturaleza recursiva del algoritmo facilita hallar una ecuación de recurrencia para el número de operaciones efectuadas. Contamos las operaciones aritméticas efectuadas con componentes de P y raíces de 1. Sean $M(k)$, $S(k)$ y $R(k)$ el número de multiplicaciones, sumas y restas, respectivamente, efectuadas por `FFTrecursiva` para calcular la transformada directa de Fourier de un 2^k-vector. Las tres operaciones se efectúan, una de cada una, en el cuerpo del ciclo **`for`**, así que $M(k) = S(k) = R(k)$. Obtendremos $M(k)$.

$$M(0) = 0$$
$$M(k) = 2^{k-1} + 2M(k-1),$$

donde el primer término del miembro derecho, es decir, 2^{k-1}, cuenta las multiplicaciones que se efectúan en el ciclo **`for`**, y el segundo término, $2M(k-1)$, las efectuadas por las invocaciones recursivas de `FFTrecursiva` para calcular los valores de los arreglos `pares` e `impares`. Es fácil ver que $M(k) = 2^{k-1}k$. Por tanto, $M(k) = S(k) = R(k) = 2^{k-1}k = (n/2)\lg n$. Puesto que las operaciones se efectúan con números complejos, este resultado deberá multiplicarse por una constante pequeña para reflejar el hecho de que cada operación con complejos requiere varias operaciones ordinarias.

El algoritmo 12.4 requeriría mucho tiempo y espacio extra para la "contabilidad" que la recursión necesita. No obstante, el desglose del polinomio parece ser lo bastante sistemático como para que sea posible establecer un esquema que efectúe el mismo cálculo "de abajo hacia arriba" sin usar un programa recursivo. El ejemplo del diagrama de árbol de la figura 12.5 deberá sugerir el patrón de los cálculos. Las profundidades en el árbol corresponden a la profundidad de la recursión, pero podríamos eliminar la recursión si iniciamos el cálculo en las hojas. Éstas son los componentes del vector P permutados de cierta manera. Determinar la permutación correcta, π_k, es la clave para construir una implementación eficiente del esquema de evaluación. Daremos π_k (y demostraremos que es correcta) después de presentar el algoritmo no recursivo. Invitamos a los lectores a tratar de determinar la manera de definir π_k antes de continuar.

■ ■ ■

Obsérvese que, en cada nivel del árbol, se calcula el mismo número de valores: 2^k, puesto que en la profundidad d hay 2^d nodos, o polinomios, que evaluar en 2^{k-d} raíces de unidad. Puesto que los valores calculados en un nivel sólo se necesitan para el cálculo de dos valores en el siguiente nivel, bastará un arreglo, `transformada`, con 2^k elementos para almacenar los resultados de los cálculos. La figura 12.6 ilustra el cálculo de dos valores en un nodo que está a la profundidad d, utilizando un valor de cada uno de sus hijos. El diagrama podría ayudar a aclarar la forma en que el algoritmo maneja los índices.

Algoritmo 12.5 Transformada rápida de Fourier

Entradas: El vector $P = (p_0, p_1, \ldots, p_{n-1})$, como arreglo **`Complex`** con n elementos, donde $n = 2^k$; y un entero $k > 0$. (Para procesar un vector de **`float`**, se le copia en la parte real del arreglo **`Complex`** P y se ponen en 0 todas las partes imaginarias.)

Salidas: El arreglo **`Complex`** `transformada`, la transformada discreta de Fourier de P, con n elementos. El arreglo se pasa como parámetro y el algoritmo lo llena.

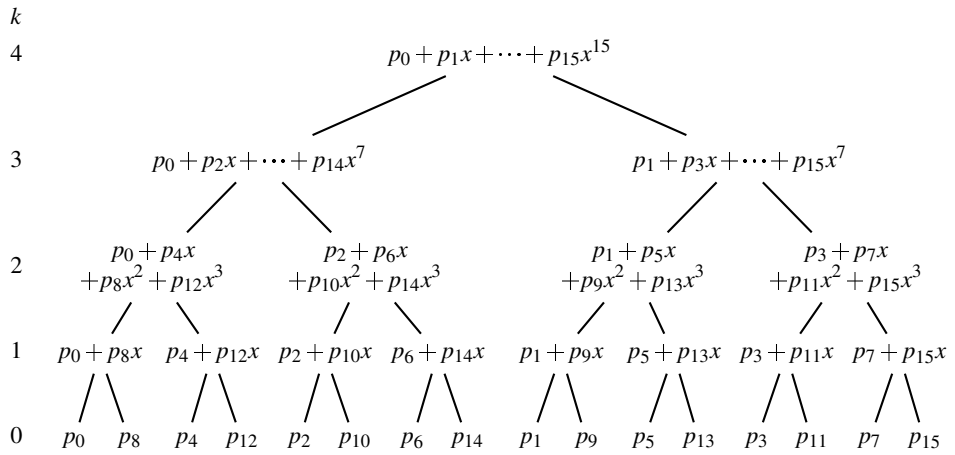


Figura 12.5 Evaluación de polinomios en raíces de unidad: para un polinomio p en cualquier nodo interno, el hijo izquierdo es p_{pares} y el hijo derecho es $p_{impares}$.

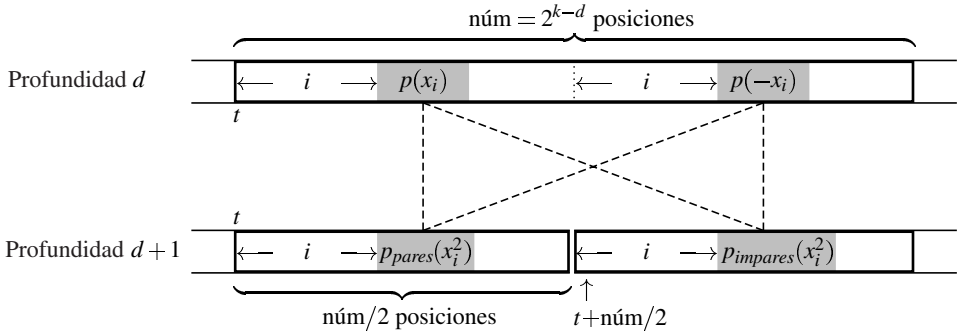


Figura 12.6 Ilustración de FFT: En el nodo que se muestra a la profundidad d, el polinomio p se evaluará en $x_0, x_1, \ldots, x_{2k-(d+1)-1}, -x_0, -x_1, \ldots, -x_{2k-(d+1)-1}$, donde $p(x_i) = p_{pares}(x_i^2) + x_i\, p_{impares}\ (x_i^2)$ y $p(-x_i) = p_{pares}\ (x_i^2) - x_i\, p_{impares}\ (x_i^2)$. El diagrama muestra qué valores de profundidades anteriores se usan para calcular $p(x_i)$ y $p(-x_i)$.

Comentarios: Suponemos que la clase **`Complex`** ofrece operaciones aritméticas con complejos, a fin de simplificar el pseudocódigo. Esta clase no existe actualmente en Java.

Suponemos que `omega` es un arreglo **`Complex`** que contiene las raíces n-ésimas de 1: ω^0, $\omega, \ldots, \omega^{n-1}$. El arreglo **`Complex`** `transformada` se inicializa de modo que contenga los valores para la profundidad $k - 1$, no las hojas, del árbol de la figura 12.5. π_k es cierta permutación de $\{0, 1, \ldots, n - 1\}$.

```
void fft(Complex[] P, int k, Complex[] transformada)
    int n = 2^k;
    int d;  // la profundidad actual en el árbol
    int num;
    // num es el número de valores que se calcularán
    // en cada nodo a la profundidad d.
    int t;
    // t es el índice en transformada que corresponde al
    // primero de esos valores para un nodo dado
    int j;
    // j cuenta los pares de valores a calcular
    // para ese nodo.
    int m;
    // m se usa como se usó en FFTrecursiva para
    // escoger el elemento correcto de omega
    Complex transPrevia;   // variable temporal


    // Para inicializar transformada, evaluamos polinomios de
    // grado 1 = 2^1 - 1 en las raíces cuadradas de 1.
    for (t = 0; t <= n - 2; t += 2)
       transformada[t] = P[π_k(t)] + P[π_k(t + 1)];
       transformada[t+1] = P[π_k(t)] - P[π_k(t + 1)];

    // El cálculo principal
    m = n/2; num = 2;
    for (d = k - 2; d >= 0; d --)
       m = m/2; num = 2 * num;
       for (t = 0; t <= (2^d - 1) * num; t += num)
           for (j = 0; j <= (num/2) - 1; j ++)
              xPImpares = omega[m*j] * transformada[t + num/2 + j];
              transPrevia = transformada[t + j];
              transformada[t + j] = transPrevia + xPImpares;
              transformada[t + num/2 + j] = transPrevia - xPImpares;
```

Un análisis del número de operaciones que `fft` efectúa da un resultado apenas distinto del que se obtiene para `FFTrecursiva`. Los enunciados que efectúan el grueso de los cálculos (una multiplicación compleja, una suma compleja y una resta compleja) están en un ciclo **`for`** triplemente anidado. Es fácil verificar que `num` $= 2^{k-d}$, así que los intervalos de los índices de los ciclos indican que el número de cada operación que se efectúa en estos enunciados es

$$\sum_{d=0}^{k-2} 2^d \frac{\texttt{num}}{2} = \sum_{d=0}^{k-2} 2^d 2^{k-d-1} = \sum_{d=0}^{k-2} 2^{k-1} = (k-1)2^{k-1} = \tfrac{1}{2}\, n(\lg(n) - 1.$$

El primer ciclo **for**, que inicializa `transformada`, efectúa $n/2$ sumas y $n/2$ restas, así que el total es $\frac{3}{2}\, n \lg(n) - \frac{1}{2}$ operaciones aritméticas de complejos. Afirmamos que la permutación π_k se puede calcular con la suficiente facilidad como para que el tiempo de ejecución de `fft` esté en $\Theta(n \log n)$.

Cabe señalar que la transformada rápida de Fourier nos permite evaluar un polinomio de grado $n - 1$ en n puntos distintos con un costo de sólo $\frac{1}{2}n(\lg(n) - 1)$ multiplicaciones de complejos. La cota inferior para la multiplicación de polinomios dada en la sección 12.2 sugiere que esto no es posible para n puntos arbitrarios. La rapidez de la FFT se debe a la forma en que utiliza algunas propiedades de las raíces de unidad.

Ahora bien, ¿qué es π_k? Sea t un entero entre 0 y $n - 1$, donde $n = 2^k$. Entonces t se puede representar en binario con $[b_0 b_1 \cdots b_{k-1}]$, donde cada b_j es 0 o 1. Sea $inv_k(t)$ el número representado por esos bits en el orden inverso, es decir, por $[b_{k-1} \cdots b_1 b_0]$. Afirmamos que $\pi_k(t) = inv_k(t)$. El lema 12.3 describe los valores calculados por la FFT utilizando $\pi_k = inv_k$. Lo usaremos en el teorema 12.4 para demostrar la corrección del algoritmo, con lo que también demostraremos que es correcto escoger esta π_k. La demostración del lema viene después del teorema. Antes, necesitamos algo de notación.

Definición 12.3

Sea P el n-vector complejo `P[0], P[1], ..., P[n-1]`, donde $n = 2^k$. Si consideramos que n y k son fijos, para toda t tal que $0 \le t \le n - 1$ y para toda d tal que $0 \le d \le k - 1$, definimos el vector de coeficientes $c_{t,d}[j]$ como

$$c_{t,d}[j] = \texttt{P}[2^d j + rev_k(t)] \qquad \text{para } 0 \le j \le 2^{k-d} - 1.$$

Ahora definimos $P_{t,d}$ como el polinomio de grado $2^{k-d} - 1$ con coeficientes $c_{t,d}[j]$. ■

Lema 12.3 Sea π_k del algoritmo 12.5 inv_k. Se cumplen las siguientes afirmaciones para $d = k - 1$ antes de que se ingrese por vez primera en el ciclo **for** triplemente anidado y para toda d tal que $k - 2 \ge d \ge 0$ al final de cada ejecución del cuerpo del ciclo **for** exterior.

1. $m = 2^d$ y `num` $= 2^{k-d}$.
2. Para $t = r2^{k-d}$ donde $0 \le r \le 2^d - 1$, `transformada[t], ..., transformada[t+num-1]` contienen los valores de $P_{t,d}$ evaluados en las raíces (2^{k-d})-ésimas de 1, donde $P_{t,d}$ se explicó en la definición 12.3.

Teorema 12.4 El algoritmo 12.5 calcula los valores de

$$p(x) = \texttt{P}[0] + \texttt{P}[1]x + \cdots + \texttt{P}[\texttt{n} - 2]x^{n-2} + \texttt{P}[\texttt{n} - 1]x^{n-1}$$

en las raíces n-ésimas de 1. Es decir, calcula la transformada discreta de Fourier de P.

Demostración Sea $d = 0$ en el lema 12.3. Entonces el único valor de t es 0 y el lema dice que `transformada[0], ..., transformada`$[2^k - 1]$ contienen los valores de $P_{0,0}$ en las raíces 2^k-ésimas de 1. Los coeficientes de $P_{0,0}$ son $c_{0,0}[j] = \texttt{P}[2^0 j + inv_k(0)] = \texttt{P}[j]$ para $0 \le j \le 2^k - 1$, así que $P_{0,0}$ es el polinomio p. □

Demostración del lema 12.3 Demostraremos el lema por inducción con d, donde d varía entre 0 y $k - 1$. Sea la base $d = k - 1$. La afirmación 1 obviamente se cumple. La afirmación 2 dice que t varía entre 0 y $2^k - 2$ (es decir, $n - 2$) en incrementos de 2 y que, para cada t, `transformada[t]` y `transformada[t+1]` contienen $P_{t,k-1}(1)$ y $P_{t,k-1}(-1)$. Sin embargo (utilizando el lema 12.5), los coeficientes de $P_{t,k-1}$ son

$$c_{t,k-1}[0] = \mathtt{P}[2^{k-1}\,0 + inv_k(t)] = \mathtt{P}[inv_k(t)],$$
$$c_{t,k-1}[1] = \mathtt{P}[2^{k-1}\,1 + inv_k(t)] = \mathtt{P}[inv_k(t+1)].$$

Es decir, $p_{t,k-1}(x) = \mathtt{p}[inv_k(\mathrm{t})] + \mathtt{P}[inv_k(t+1)]x$. Esto corresponde exactamente a los valores iniciales que se asignan a `transformada` en el primer ciclo **`for`**.

Supóngase ahora que $0 \le d < k - 1$ y que las afirmaciones 1 y 2 se cumplen para $d + 1$. Es fácil deducir que la afirmación 1 se cumple para d. Cabe señalar que w^{2^d} y $w^{2^{d+1}}$ son (2^{k-d})-ésimas y raíces $(2^{k-(d+1)})$-ésimas primitivas de 1, respectivamente. Aplicando la hipótesis inductiva vemos que, para $0 \le i \le \mathtt{num}/2 - 1$, el cuerpo del ciclo **`for`** triplemente anidado calcula

$$\mathtt{xPImpares} = (w^{2^d})^i\, P_{t+\mathrm{num}/2,\,d+1}\left((w^{2^{d+1}})^i\right),$$
$$\mathtt{transformada}[t+i] = P_{t,\,d+1}\left((w^{2^{d+1}})^i\right) + (w^{2^d})^i + P_{t+\mathrm{num}/2,\,d+1}\left((w^{2^{d+1}})^i\right),$$
$$\mathtt{transformada}[t+\mathrm{num}/2+i] = P_{t,\,d+1}\left((w^{2^{d+1}})^i\right) - (w^{2^d})^i\, P_{t+\mathrm{num}/2,d+1}\left((w^{2^{d+1}})^i\right).$$

Por tanto, $P_{t,\,d}(x) = P_{t,\,d+1}(x^2) + x\,P_{t+\mathrm{num}/2,d+1}(x^2)$ y los elementos de arreglo `transformada[0]`, …, `transformada[t+num-1]` contienen $P_{t,d}$ evaluado en

$$(w^{2^d})^0,\ w^{2^d},\ \ldots,\ (w^{2^d})^{2k-d-1},\ -(w^{2^d})^0,\ -w^{2^d},\ \ldots,\ -(w^{2^d})^{2k-d-1},$$

es decir, en las raíces (2^{k-d})-ésimas de 1. Los coeficientes de P_{t-d} se obtienen como sigue para $0 \le j \le 2^{k-d} - 1$:

$$c_{t,d}[j] = \begin{cases} c_{t,\,d+1}[j/2] & \text{si } j \text{ es impar.} \\ c_{t+\mathrm{num}/2,\,d+1}[(j-1)/2] & \text{si } j \text{ es par.} \end{cases}$$

Por consiguiente, aplicando la hipótesis inductiva, si j es par,

$$c_{t,d}[j] = c_{t,d+1}[j/2] = \mathtt{P}[2^{d+1}(j/2) + inv_k(t)] = \mathtt{P}[2^d j + inv_k(t)]$$

tal como se requiere. Si j es impar,

$$_{ct,d}[j] = c_{t+\mathrm{num}/2,\,d+1}[(j-1)/2] = \mathtt{P}[2^{d+1}(j-1)/2 + inv_k(t + 2^{k-(d+1)})]$$
$$= (\text{por el lema 12.5})\ \mathtt{P}[2^d(j-1) + inv_k(t) + 2^d] = \mathtt{P}[2^d j + inv_k(t)],$$

que también cumple con los requisitos. □

En la demostración usamos el lema siguiente, cuya demostración se deja como ejercicio.

Lema 12.5 Para $k, a, b > 0$, $b \leq k$ y $a + 2^{k-b} < 2^k$, si a es un múltiplo de 2^{k-b+1}, entonces $inv_k(a + 2^{k-b}) = inv_k(a) + 2^{b-1}$. □

12.4.2 Convolución

Para justificar el cálculo de la convolución, examinaremos el problema de la multiplicación simbólica de polinomios. Supóngase que nos dan los vectores de coeficientes

$$P = (p_0, p_1, \ldots, p_{m-1}),$$
$$Q = (q_0, q_1, \ldots, q_{m-1}),$$

para los polinomios $p(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1x + p_0$ y $q(x) = q_{m-1}x^{m-1} + q_{m-2}x^{m-2} + \cdots + q_1x + q_0$. El problema consiste en hallar el vector $R = (r_0, r_1, \ldots, r_{2m-1})$ de coeficientes del polinomio producto $r(x) = p(x)q(x)$. Los coeficientes de r están dados por la fórmula

$$r_i = \sum_{j=0}^{i} P_j q_{i-j} \quad \text{para } 0 \leq i \leq 2m - 1$$

donde p_k y q_k se toman como cero si $k > m - 1$. (Cabe señalar que $r_{2m-1} = 0$ ya que r es de grado $2m - 2$; lo incluimos por comodidad.) R se parece mucho a la convolución de P y Q. Sean $\vec{P}$ y $\vec{Q}$ los $2m$-vectores que se obtienen al añadir m ceros a P y Q, respectivamente. Entonces, $R = \vec{P} \star \vec{Q}$. Así pues, nuestra investigación de la multiplicación de polinomios deberá dar pie a un algoritmo de convolución.

Consideremos el bosquejo siguiente para la multiplicación de polinomios:

1. Evaluar $p(x)$ y $q(x)$ en $2m$ puntos: $x_0, x_1, \ldots, x_{2m-1}$.
2. Multiplicar punto por punto para determinar los valores de $r(x)$ en esos $2m$ puntos; es decir, calcular $r(x_i) = p(x_i)q(x_i)$ para $0 \leq i \leq 2m - 1$.
3. Determinar los coeficientes del polinomio único de grado $2m - 2$ que pasa por los puntos $\{(x_i, r(x_i), \mid 0 \leq i \leq 2m - 1\}$. (Un teorema muy conocido dice que pueden determinarse los coeficientes de un polinomio de grado d si se conocen los valores del polinomio en $d + 1$ puntos.)

Si los puntos $x_0, \ldots, x_{2m-1}$ se escogieran al azar, el método que bosquejamos requeriría mucho más trabajo que un cálculo directo de $\vec{P} \star \vec{Q}$, pero la FFT puede evaluar p y q en las raíces $(2m)$-ésimas de 1 de forma muy eficiente (suponiendo que m es una potencia de 2). Por tanto, el paso 1 puede ejecutarse en tiempo $\Theta(m \log m)$. El paso 2 sólo requiere $2m$ multiplicaciones. ¿Cómo llevamos a cabo el paso 3?

Sea ω una raíz $(2m)$-ésima primitiva de 1 y, para $0 \leq j \leq 2m - 1$, sea $w_j = r(\omega^j) = p(\omega^j)q(\omega^j)$. Podemos determinar los coeficientes de r resolviendo el conjunto siguiente de ecuaciones lineales simultáneas para $r_0, r_1, \ldots, r_{2m-1}$.

$$\begin{aligned} r_0 + r_1\omega^0 + \cdots + r_{2m-2}(\omega^0)^{2m-2} + r_{2m-1}(\omega^0)^{2m-1} &= w_0, \\ r_0 + r_1\omega + \cdots + r_{2m-2}(\omega)^{2m-2} + r_{2m-1}(\omega)^{2m-1} &= w_1, \\ \vdots & \\ r_0 + r_1\omega^{2m-1} + \cdots + r_{2m-2}(\omega^{2m-1})^{2m-2} + r_{2m-1}\,(\omega^{2m-1})^{2m-1}) &= w_{2m-1}. \end{aligned} \tag{12.5}$$

Si r se hubiera evaluado en $2m$ puntos arbitrarios, se podría haber usado un algoritmo $\Theta(m^3)$, como la eliminación gaussiana, para resolver las ecuaciones. Una vez más, aprovechamos el hecho de que los puntos son raíces de unidad para obtener un algoritmo $\Theta(m \log m)$. Las fórmulas de la ecuación (12.5) se pueden escribir como ecuación matricial $F_{2m}R = W$, donde F_{2m} se explicó en la definición 12.2 y W es el vector $(w_0, w_1, \ldots, w_{2m-1})$. Así pues,

$$\vec{P} \star \vec{Q} = R = F_{2m}^{-1} = F_{2m}^{-1}\,(F_{2m}\vec{P} * F_{2m}\vec{Q}),$$

donde $\star$ denota convolución y $*$ denota multiplicación de componentes. Quedan tres problemas: demostrar que F_n sí puede invertirse para toda $n > 0$, demostrar que la fórmula $U \star V = F_n^{-1}(F_nU \times F_nV)$ se cumple para n-vectores arbitrarios U y V, y hallar una forma eficiente de calcular la transformada inversa. La fórmula para calcular $U \star V$ no es consecuencia directa de la fórmula para calcular R porque $\vec{P}$ y $\vec{Q}$ tienen la propiedad de que la mitad de sus componentes son cero.

Lema 12.6 Para $n > 0$, F_n es invertible y el (i, j)-ésimo elemento de su inversa es $(1/n)\omega^{-ij}$ para $0 \le i, j \le n - 1$.

Demostración Sea $\widetilde{F}_n$ la matriz que el lema afirma es F_n^{-1}. Demostraremos que $F_n\widetilde{F}_n = I$; $\widetilde{F}_nF_n = I$ por un argumento similar.

$$(F_n\,\widetilde{F}_n)_{ij} = \sum_{k=0}^{n-1} \omega^{ik}\,\frac{1}{n}\,\omega^{-kj} = \frac{1}{n}\sum_{k=0}^{n-1}(\omega^{i-j})^k.$$

En el caso de elementos que no están en la diagonal (es decir, $i \neq j$), $(F_n\widetilde{F}_n)_{ij} = 0$ por la propuesta 12.8 para raíces de unidad dado que $0 < |i - j| < n$ (véase la sección 12.4.3). Para elementos de la diagonal (o sea, para $i = j$),

$$(F_n\widetilde{F}_n)_{ij} = \frac{1}{n}\sum_{k=0}^{n-1}(\omega^0)^k = \frac{1}{n}\sum_{k=0}^{n-1} 1 = 1. \qquad \square$$

Teorema 12.7 Sean U y V n-vectores. Entonces $U \star V = F_n^{-1}\,(F_nU * F_nV)$, donde $\star$ denota convolución y $*$ denota multiplicación de componentes.

Demostración Demostraremos que $F_n(U \star V) = F_nU * F_nV$. Para $0 \le i \le n - 1$, el i-ésimo componente de $F_nU * F_nV$ es

$$\left(\sum_{j=0}^{n-1}\omega^{ij}\,u_j\right)\left(\sum_{k=0}^{n-1}\omega^{ij}\,v_k\right) = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1}u_jv_k\omega^{i(j+k)}.$$

El t-ésimo componente de $U \star V$ es $\sum_{j=0}^{n-1} u_j v_{t-j}$ donde los subíndices se toman módulo n. Por tanto, el i-ésimo componente de $F_n(U \star V)$ es

$$\sum_{t=0}^{n-1} \left(\omega^{it} \sum_{j=0}^{n-1} u_j v_{t-j} \right) = \sum_{j=0}^{n-1} \sum_{t=0}^{n-1} u_j v_{t-j} \omega^{it}.$$

Sea $k = t - j \pmod{n}$ en la sumatoria interior. Para cada j, puesto que t varía entre 0 y $n - 1$, k también variará de 0 a $n - 1$, aunque en diferente orden. También, para cualquier p, $\omega^p = \omega^{p \bmod n}$, así que el i-ésimo componente de $F_n(U \star V)$ es

$$\sum_{j=0}^{n-1} \sum_{k=0}^{n-1} u_j v_k \omega^{j(j+k)}$$

que es exactamente el i-ésimo componente de $F_n U * F_n V$. □

El lema 12.6 indica que la matriz F_n^{-1} no es muy distinta de F_n. Los elementos de F_n^{-1} son ω^{-ij}. Sus filas son las filas de F_n dispuestas en un orden distinto. Específicamente, dado que ω^{n-i}, para $1 \leq i \leq n - 1$ la i-ésima fila de F_n es la $(n - i)$-ésima fila de nF_n^{-1}. La fila 0 es igual en ambas matrices. Por tanto, la transformada discreta inversa de Fourier de un n-vector A se puede calcular como sigue.

Algoritmo 12.6 Transformada discreta inversa de Fourier

Entradas: El vector $A = (a_0, a_1, \ldots, a_{n-1})$, como arreglo **Complex** de n elementos, donde $n = 2^k$; y el entero $k > 0$.

Salidas: El vector **Complex** $B = (b_0, b_1, \ldots, b_{n-1})$, la transformada discreta inversa de Fourier de A; es decir, $B = F_n^{-1} A$. El arreglo se pasa como parámetro y el algoritmo lo llena.

Comentarios: Los comentarios acerca del algoritmo 12.5 también son válidos aquí.

```
void FFTinversa(Complex[] A, int k, Complex[] B)
    int n = 2^k;
    int i;
    Complex[] transformada = new Complex[n];
    fft(A, k, transformada);
    B[0] = transformada[0] / n;
    for (i = 1; i <= n - 1; i ++)
        B[i] = transformada[n - 1] / n;
```

Análisis

La FFT efectúa $\frac{1}{2}n(\lg(n) - 1)$ multiplicaciones de complejos (y el mismo número de sumas y restas de complejos), así que el algoritmo 12.6 ejecuta $\frac{1}{2}n(\lg(n) + 1)$ pasos $*/$ con complejos, y ambos se ejecutan en tiempo $\Theta(n \log n)$. El cálculo de la convolución de dos n-vectores utilizando la FFT tarda un tiempo $\Theta(n \log n)$.

12.4.3 Apéndice: números complejos y raíces de unidad

El campo **C** de los números complejos se obtiene uniendo i, la raíz cuadrada de -1, con el campo de los números reales **R**. Por tanto, $\mathbf{C} = \mathbf{R}(i) = \{a + bi \mid a, b \in \mathbf{R}\}$. Si $z = a + bi$, decimos que a es la parte real de z y b es la parte imaginaria. Sean $z_1 = a_1 + b_1 i$ y $z_2 = a_2 + b_2 i$. Entonces, por definición,

$$
\begin{aligned}
z_1 + z_2 &= (a_1 + a_2) + (b_1 + b_2)i, \\
z_1 z_2 &= (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i, \\
\frac{1}{z_1} &= \frac{a_1}{a_1^2 + b_1^2} - \frac{b_1 i}{a_1^2 + b_1^2} \qquad \text{para } z_1 \neq (0 + 0i).
\end{aligned}
$$

Es fácil definir la división y la resta utilizando las ecuaciones anteriores.

Detalle de Java: Para simplificar la representación de números complejos y sus operaciones en el pseudocódigo de los algoritmos, supondremos que existe una clase **`Complex`** que permite las operaciones aritméticas $*$, $/$, $+$ y $-$; tal clase no existe actualmente en Java, pero C++ y Fortran reconocen esta notación. Es fácil definir una clase con dos campos de ejemplar, `re` e `im`, ambos de tipo **`float`** o **`double`**. (Las abreviaturas `re` e `im` se usan en muchos textos de matemáticas para denotar las partes real e imaginaria de un número complejo.) Sin embargo, es posible que el programador tenga que definir las operaciones aritméticas como funciones (métodos estáticos) y usar notación funcional en el código real, en vez de notación de operadores.

Un número complejo se puede representar como un vector en un plano utilizando las partes real e imaginaria como coordenadas cartesianas. Es más fácil ver la interpretación geométrica de la multiplicación de números complejos si usamos coordenadas polares r y θ, donde r es la longitud del vector y θ es el ángulo (medido en radianes) que subtiende con el eje horizontal, o real. (Véase la figura 12.7.) El producto de dos números complejos (r_1, θ_1) y (r_2, θ_2) es $(r_1 r_2, \theta_1 + \theta_2)$. En la figura 12.8 se da un ejemplo.

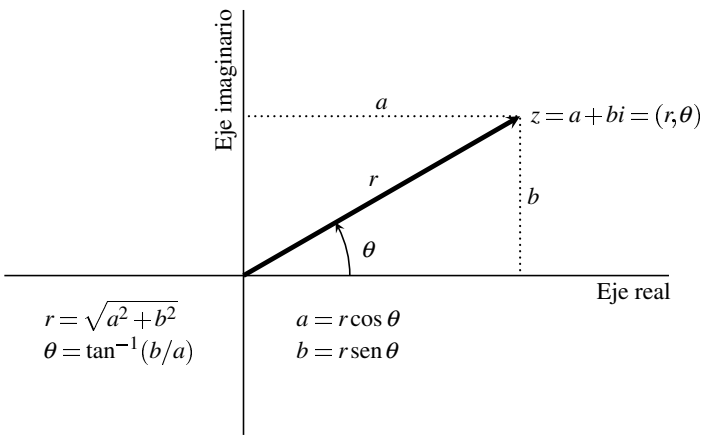


Figura 12.7 Coordenadas cartesianas y polares para números complejos

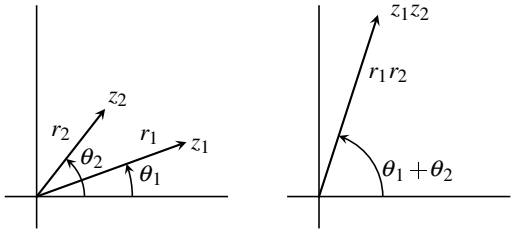


Figura 12.8 Multiplicación de números complejos: las magnitudes r_1 y r_2 se multiplican; los ángulos θ_1 y θ_2 se suman.

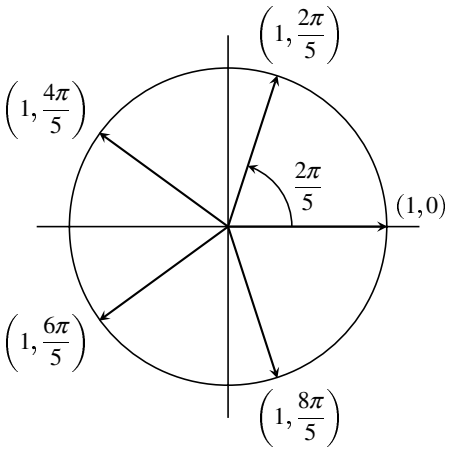


Figura 12.9 Raíces quintas de unidad (coordenadas polares)

El campo complejo **C** está *cerrado algebraicamente*. Esto implica que todo polinomio de grado n con coeficientes en **C** tiene n raíces (no necesariamente distintas). Por tanto, $x^n - 1$ tiene n raíces, que se denominan *raíces n-ésimas de unidad*, o *raíces n-ésimas de* 1. Las coordenadas polares de 1 son (1, 0). Para determinar una raíz (r, θ) de $x^n - 1$ resolvemos la ecuación $(r^n, n\theta) = (1, 0)$. Puesto que r es real y no negativo, r debe ser 1, así que todas las raíces de unidad se representan con vectores de longitud unitaria. Dado que $n\theta = 0$, $\theta = 0$ y hemos descubierto que (1, 0) —es decir, 1— es una solución, lo cual no es gran sorpresa. Para hallar las otras raíces de unidad aprovechamos el hecho de que un ángulo de 0 radianes equivale a un ángulo de $2\pi j$ radianes para cualquier entero j. Las n raíces distintas son $\{1, 2\pi j/n) \mid 0 \leq j \leq n - 1\}$. Los vectores que representan estos números cortan el círculo unitario en n sectores circulares iguales como se muestra en la figura 12.9.

Si ω es una raíz n-ésima de 1, entonces ω^k también es una raíz n-ésima de 1, puesto que $(\omega^k)^n = \omega^{nk} = (\omega^n)^k = 1^k = 1$. Si ω es una raíz n-ésima de 1 y 1, ω, ω^2, …, ω^{n-1} son todas distintas,

decimos que ω es una *raíz n-ésima primitiva de unidad*. Una raíz n-ésima primitiva de unidad es $(1, 2\pi/n) = \cos(2\pi/n) + i\,\text{sen}(2\pi/n)$. Las propiedades siguientes se usan en la sección 12.4.

Propuesta 12.8 Para $n \geq 2$, la suma de todas las raíces n-ésimas de 1 es cero. También, si ω es una raíz n-ésima primitiva de 1 y c es un entero no divisible entre n, entonces $\sum_{j=0}^{n-1} (\omega^c)^j = 0$.

Demostración Sea ω una raíz n-ésima primitiva de 1. Entonces $\omega^0, \omega, \omega^2, \ldots, \omega^{n-1}$ son todas raíces n-ésimas de 1. Su suma es

$$\sum_{j=0}^{n-1} \omega^j = \frac{\omega^n - 1}{\omega - 1} = \frac{1 - 1}{\omega - 1} = 0.$$

La segunda afirmación se demuestra de forma similar. □

Propuesta 12.9 Si n es par y ω es una raíz n-ésima primitiva de 1, entonces

1. ω^2 es una raíz $(n/2)$-ésima primitiva de 1, y
2. $\omega^{n/2} = -1$.

Demostración La demostración se deja como ejercicio. □

Ejercicios

Sección 12.2 Evaluación de funciones polinómicas

12.1 Cualquier polinomio $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ se puede factorizar a

$$p(x) = a_n(x - r_1)(x - r_2) \cdots (x - r_n),$$

donde $r_1, \ldots, r_n$ son las raíces de p. ¿Podría usarse esta factorización como base de un algoritmo para evaluar $p(x)$? ¿Cómo, o por qué no?

12.2 Afirmamos que un algoritmo para evaluar un polinomio de grado n debe efectuar al menos n multiplicaciones y/o divisiones en el peor caso. En casos especiales, podríamos lograr algoritmos con mejor desempeño. Idee un algoritmo rápido para evaluar cada uno de los polinomios siguientes. Las entradas son n y x.

a. $p(x) = x^n + x^{n-1} + \cdots + x + 1$.
¿Cuántas operaciones aritméticas lleva a cabo su algoritmo?

b. $p(x) = \sum_{k=0}^{n} \binom{n}{k} x^k$, donde $\binom{n}{k}$ son los coeficientes binomiales (ecuación 1.1).
¿Cuántas operaciones aritméticas efectúa su algoritmo?

12.3 Escriba la factorización que se usaría para evaluar $p(x) = x^7 + 6x^6 - 7x^5 + 12x^4 + 2x^2 - 3x - 8$ por

a. el método de Horner.

b. procesamiento previo de coeficientes.

12.4 ¿Qué parte(s) de las demostraciones del teorema 12.2 y/o el lema 12.1 no funcionarían si se permitiera dividir?

12.5 ¿Cómo debe modificarse o qué debe añadirse al procedimiento para evaluar polinomios con procesamiento previo de coeficientes para que funcione con polinomios no mónicos? ¿Cuántas multiplicaciones y/o divisiones efectúa el algoritmo extendido?

12.6 Suponga que $A(1) = 1$ y que, para $k > 1$, $A(k) = 2A(k - 1) + 2$. Muestre que la solución de esta ecuación de recurrencia es $A(k) = (3n - 1)/2$, donde $n = 2^k - 1$.

12.7 Utilizando la terminología del primer párrafo de la sección 12.2.3, determine I, I', A_I y el algoritmo que da A_I a partir de I para el problema de evaluar un polinomio con procesamiento previo de coeficientes usando el método descrito en la sección 12.2.3.

Sección 12.3 Multiplicación de vectores y matrices

12.8 En la sección 12.3.1 dijimos que calcular el producto punto $U \cdot V$ de dos n-vectores con componentes reales requiere por lo menos n pasos $*/$. ¿Cuántos pasos $*/$ se requieren si U siempre tiene componentes enteros? ¿Por qué?

12.9 Calcule el número exacto de multiplicaciones y sumas que efectúa el algoritmo 12.3 cuando n es impar.

12.10 Sean A y B matrices de $n \times n$ que se van a multiplicar, y suponga que es necesario traer un elemento de matriz de la memoria cada vez que se usa en el cálculo. ¿Cuántas veces se trae cada elemento de A y B para calcular AB

a. con el algoritmo acostumbrado?

b. con el algoritmo de Winograd (si n es par)?

**** 12.11**

a. Demuestre que el algoritmo de Strassen, utilizando la cuarta modificación descrita hacia el final de la sección 12.3.4, efectúa menos de $4.7n^{\lg 7}$ operaciones aritméticas con los elementos de las matrices, sea o no n una potencia de 2.

b. Muestre cómo reducir a 15 las 18 sumas dadas para una invocación del algoritmo de Strassen.

Sección 12.4 La transformada rápida de Fourier y convolución

12.12

a. ¿Por qué se necesitan las restricciones "$n \geq 2$" y "c no es divisible entre n" en la propuesta 12.8?

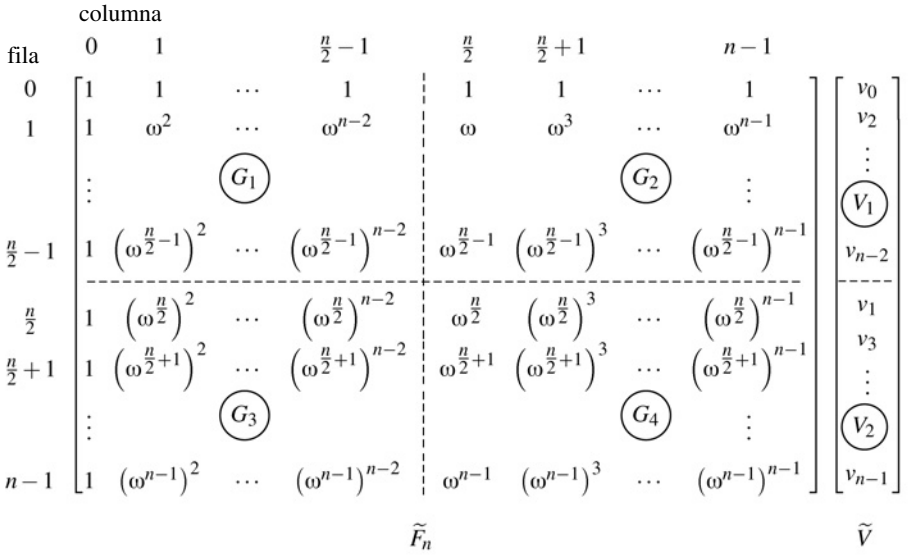


Figura 12.10 Matriz y vector para el ejercicio 12.16

b. Demuestre la propuesta 12.9 para raíces de unidad.

12.13 Sea $p(x) = p_7x^7 + p_6x^6 + \cdots + p_1x + p_0$. Ejecute los pasos de la FFT con p para mostrar cómo evalúa p en las raíces octavas de 1: 1, ω, i, $i\omega$, -1, $-\omega$, $-i$, $-i\omega$.

12.14 Suponga que le dan las partes reales e imaginarias de dos números complejos. Demuestre que las partes real e imaginaria de su producto se pueden calcular con sólo tres multiplicaciones.

12.15 Demuestre el lema 12.5.

★ **12.16** Sea $n = 2^k$ para alguna $k > 0$, sea ω una raíz n-ésima primitiva de 1, y sea F_n como en la definición 12.2. Sea V un n-vector complejo. Este problema describe la FFT (recursivamente) desde el punto de vista del producto matriz-vector F_nV, no como una evaluación de polinomios. Observe la correspondencia de diversos pasos de este algoritmo con pasos de `FFTrecursiva`.

Sea $\widetilde{F}_n$ la matriz de $n \times n$ que se obtiene a partir de F_n colocando todas las columnas de índice par antes de las columnas de índice impar. (Cabe señalar que no se trata de la $\widetilde{F}_n$ que apareció en la demostración del lema 12.6.) Tenga $\widetilde{V}$ todos los componentes de V que tienen índice par antes de todos los componentes de V que tienen índice impar. Es decir, para $0 \le j \le n/2 - 1$, $\widetilde{f}_{ij} = \omega^{i(2j)}$, $\widetilde{f}_{i,j+n/2} = \omega^{i(2j+1)}$, $\widetilde{v}_j = v_{2j}$ y $\widetilde{v}_{j+n/2} = v_{2j+1}$. Divida $\widetilde{F}_n$ en cuatro matrices de $(n/2) \times (n/2)$ llamadas G_1, G_2, G_3 y G_4, y divida $\widetilde{V}$ en dos $(n/2)$-vectores llamados V_1 y V_2 como se muestra en la figura 12.10. Ahora

$$\widetilde{F}_n\widetilde{V} = \begin{bmatrix} G_1V_1 + G_2V_2 \\ G_3V_1 + G_4V_2 \end{bmatrix}.$$

Demuestre las afirmaciones siguientes.

a. $F_V = \widetilde{F}_n\widetilde{V}$.

b. $G_1 = G_3$.

c. $G_4 = -G_2$.

d. $G_2 = DG_1$, donde D es una matriz diagonal de $(n/2) \times (n/2)$ (es decir, todos los elementos que no están en la diagonal son cero) con $d_{ii} = \omega^i$ para $0 \leq i \leq (n/2) - 1$.

e. G_1 tiene elementos $g_{ij} = \gamma^{ij}$, donde γ es una raíz $(n/2)$-ésima primitiva de 1. Por tanto, $G_1 = F_{n/2}$, una matriz de transformada discreta de Fourier para $(n/2)$-vectores, y

$$F_nV = \begin{bmatrix} F_{n/2}V_1 + DF_{n/2}V_2 \\ F_{n/2}V_1 - DF_{n/2}V_2 \end{bmatrix}.$$

Es decir, el cálculo puede efectuarse calculando recursivamente la transformada discreta de Fourier de V_1 y de V_2, ambos $(n/2)$-vectores.

f. Deduzca ecuaciones de recurrencia para el número de multiplicaciones, sumas y restas que efectúa el algoritmo aquí descrito. Sea $\vec{D} = (1, \omega, \ldots, \omega^{(n/2)-1})$. (Cabe señalar que el producto $D(F_{n/2}V_2)$ se puede calcular como un producto de componentes de $\vec{D}$ por $F_{n/2}V_2$, lo que requiere $n/2$ multiplicaciones.) Compare sus ecuaciones de recurrencia con las que se obtuvieron para `FFTrecursiva`.

Problemas adicionales

12.17 Observe que los números de Fibonacci (ecuación 1.13) satisfacen la ecuación matricial siguiente para $n \geq 2$:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = A\begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} \qquad \text{donde } A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Entonces

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = A\begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = A^2\begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} = \cdots = A^{n-1}\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = A^{n-1}\begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

¿Cuántas operaciones aritméticas se efectúan si F_n se calcula empleando esta fórmula:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = A^{n-1}\begin{bmatrix} 1 \\ 0 \end{bmatrix}?$$

Compare este método con los algoritmos recursivo e iterativo para calcular números de Fibonacci.

Programas

1. Escriba y depure subrutinas eficientes para el algoritmo de multiplicación de matrices de Winograd y para el algoritmo acostumbrado. ¿Cuántas instrucciones ejecuta cada programa para multiplicar dos matrices $n \times n$?
2. Implemente la FFT (algoritmo 12.5). Haga que el cálculo de inv_k y el resto de la "contabilidad" sean lo más eficientes que se pueda.

Notas y referencias

Las cotas inferiores dadas en las secciones 12.2 y 12.3 para la evaluación de polinomios, con o sin procesamiento previo de coeficientes, y para la obtención de productos vector-matriz se establecieron en Pan (1966), Reingold y Stocks (1972) y Winograd (1970). El algoritmo de multiplicación de matrices de Winograd también aparece en el último trabajo. Las demostraciones de Winograd utilizan teoría de campos. Reingold y Stocks usan argumentos más sencillos como el de la demostración del teorema 12.2.

El algoritmo de Strassen para multiplicar matrices se presenta en Strassen (1969), un artículo corto que no explica cómo el autor descubrió sus fórmulas. Una mejora que efectúa 15 sumas en lugar de 18, la cual fue tema del ejercicio 12.11(b), se da en Aho, Hopcroft y Ullman (1974), donde se atribuye a S. Winograd. Se dan algunos detalles adicionales en Gonnet y Baeza-Yates (1991). El método en $O(n^{2.376})$ está en Coppersmith y Winograd (1987). Varios problemas de matrices que se pueden reducir a multiplicación y por tanto tienen soluciones en $O(n^{2.376})$ se describen en Aho, Hopcroft y Ullman (1974).

Se presentan versiones de la transformada rápida de Fourier en Cooley y Tukey (1965) y en Aho, Hopcroft y Ullman (1974). Brigham (1974) es un libro acerca de la FFT. Press, Flannery, Teukolsky y Vettering (1988) analizan exhaustivamente la teoría y aspectos de implementación de la FFT. Aho, Hopcroft y Ullman (1974) presentan una aplicación de la FFT a la multiplicación de enteros. (La cadena de dígitos $d_n d_{n-1} \cdots d_1 d_0$ que representa un entero en base b es un polinomio $\sum_{i=0}^{n} d_i b^i$.) Hay muchas otras referencias acerca de la FFT, pues se le usa ampliamente.

13

Problemas $\mathcal{NP}$-completos

13.1 Introducción

En los capítulos anteriores hemos estudiado una amplia variedad de problemas y algoritmos. Algunos de los algoritmos son directos, mientras que otros son complicados y sutiles, pero prácticamente todos ellos tienen complejidad en $O(n^3)$, donde n es el tamaño de las entradas debidamente definido. Desde el punto de vista adoptado en este capítulo, consideraremos que todos los algoritmos que hemos estudiado hasta ahora requieren un tiempo relativamente corto. Examinemos otra vez la tabla 1.1. Ahí vimos que los algoritmos cuya complejidad se describe con funciones polinómicas simples se pueden ejecutar con entradas relativamente grandes en un tiempo razonable. La última columna de la tabla muestra que si la complejidad es 2^n, el algoritmo de nada sirve a menos que las entradas sean muy pequeñas. En este capítulo nos ocuparemos de problemas cuya complejidad podría describirse con funciones exponenciales, problemas cuya resolución incluso con los mejores algoritmos conocidos requeriría muchos años o siglos de tiempo de computadora con entradas moderadamente grandes. Presentaremos definiciones encaminadas a distinguir entre los problemas *dóciles* (es decir, "no tan difíciles") que ya hemos visto y los *renuentes* (es decir, "difíciles" o muy tardados). Estudiaremos una importante clase de problemas que tienen una propiedad irritante: ni siquiera sabemos si se pueden resolver de manera eficiente o no. No se han descubierto algoritmos razonablemente rápidos para estos problemas, pero tampoco se ha podido demostrar que los problemas requieren mucho tiempo. Dado que muchos de estos problemas son problemas de optimización que se presentan con frecuencia en aplicaciones, la falta de algoritmos eficientes tiene importancia real.

13.2 $\mathcal{P}$ y $\mathcal{NP}$

En este capítulo, "$\mathcal{P}$" es una clase de problemas que se pueden resolver en "tiempo polinómico". La descripción de "$\mathcal{NP}$" es más complicada. Antes de entrar en las definiciones y teoremas formales, describiremos varios problemas que usaremos como ejemplos en todo este capítulo. Luego daremos definiciones de $\mathcal{P}$ y $\mathcal{NP}$.

13.2.1 Problemas de decisión

Muchos de los problemas que describiremos en este capítulo se dan naturalmente como problemas de optimización (aunque se les llama *problemas de optimización combinatoria*), pero también se pueden formular como *problemas de decisión*. Las clases $\mathcal{P}$ y $\mathcal{NP}$, que se definirán en las subsecciones siguientes, son clases de problemas de decisión. Básicamente, un *problema de decisión* es una pregunta que tiene dos posibles respuestas, *sí* y *no*. La pregunta se refiere a alguna *entrada*. Un *ejemplar de problema* es la combinación del problema y una entrada específica. Por lo regular, el planteamiento de un problema de decisión tiene dos partes:

1. La parte de *descripción de ejemplar* define la información que cabe esperar en la entrada.
2. La parte de *pregunta* plantea la pregunta tipo "sí o no" en sí; la pregunta contiene variables definidas en la descripción de ejemplar.

La salida de un problema de decisión es *sí* o bien *no* según sea la respuesta correcta de la pregunta, aplicada a una entrada dada. Por ello, podemos ver de manera abstracta un problema de decisión como una correspondencia entre todas las entradas y el conjunto {*sí*, *no*}.

Para entender por qué es importante una descripción precisa de las entradas, consideremos estos dos problemas:

1. *Ejemplar*: un grafo no dirigido $G = (V, E)$.
   Pregunta: ¿G contiene una camarilla de k vértices? (Una camarilla es un subgrafo completo: cualesquier dos vértices del subgrafo están unidos por una arista.)
2. *Ejemplar*: un grafo no dirigido $G = (V, E)$ y un entero k.
   Pregunta: ¿G contiene una camarilla de k vértices?

La pregunta es la misma en ambos problemas, pero en el primero k no forma parte de las entradas, así que no varía de un ejemplar a otro; dicho de otro modo, k es una *constante*. Sucede que esta pregunta se puede contestar con un algoritmo que se ejecuta en $O(k^2n^k)$. Si se considera que k es constante, el algoritmo se ejecuta en tiempo polinómico. En la segunda pregunta, k es parte de la entrada, así que es una variable. El algoritmo se sigue ejecutando en tiempo $O(k^2n^k)$, pero esta expresión no es un polinomio porque el exponente de n es una variable.

13.2.2 Algunos problemas de ejemplo

He aquí algunos problemas que estudiaremos en el capítulo. En algunos casos el problema es una simplificación o abstracción de un problema que se presenta en aplicaciones realistas. Es común simplificar los problemas difíciles en un intento por lograr algún avance y entenderlos mejor, con la esperanza de que tal entendimiento ayude a resolver el problema original.

Definición 13.1 Coloreado de grafos y número cromático

Un *coloreado* de un grafo $G = (V, E)$ es una correspondencia $C : V \to S$, donde S es un conjunto finito (de "colores") tal que si $vw \in E$, entonces $C(v) \neq C(w)$; dicho de otro modo, no se asigna el mismo color a vértices adyacentes.

El *número cromático* de G, denotado por $\chi(G)$, es el menor número de colores que se necesitan para colorear G (es decir, el k más pequeño tal que exista un coloreado C para G y $|C(V)| = k$). ■

Problema 13.1 Coloreado de grafos

Se nos da un grafo no dirigido $G = (V, E)$ para colorear.

Problema de optimización: Dado G, determinar $\chi(G)$ (y producir un coloreado óptimo, es decir, uno que sólo use $\chi(G)$ colores).

Problema de decisión: Dado G y un entero positivo k, ¿existe un coloreado de G que use cuando más k colores? (Si lo hay, decimos que G es *k-coloreable*.)

El problema de coloreado de grafos es una abstracción de ciertos tipos de problemas de calendarización. Por ejemplo, supóngase que los exámenes finales de una universidad se van a calendarizar durante una semana con tres horas de examen por día, para un total de 15 horarios. Los exámenes de algunos cursos, digamos Cálculo 1 y Física 1, se deben efectuar en horarios distintos porque muchos estudiantes están tomando ambos cursos. Sea V el conjunto de cursos, y sea E el conjunto de pares de cursos cuyos exámenes no se deben efectuar al mismo tiempo. Entonces, los exámenes se podrán calendarizar en los 15 horarios sin conflictos si y sólo si el grafo $G = (V, E)$ se puede colorear con 15 colores. ■

Problema 13.2 Calendarización de trabajos con castigos

Supóngase que hay n trabajos $J_1, \ldots, J_n$ que deben ejecutarse uno por uno. Se nos dan los tiempos de ejecución $t_1, \ldots, t_n$, los plazos $d_1, \ldots, d_n$ (medidos a partir de la fecha de inicio del primer trabajo ejecutado) y castigos por no terminar un trabajo dentro del plazo correspondiente, $p_1, \ldots, p_n$. Supóngase que los tiempos de ejecución, plazos y castigos son enteros positivos. Un *calendario* para los trabajos es una permutación π de $\{1, 2, \ldots, n\}$, donde $J_{\pi(1)}$ es el trabajo que se efectúa primero, $J_{\pi(2)}$ es el que se ejecuta en segundo lugar, y así.

Para un calendario dado, el castigo para el j-ésimo trabajo se denota con P_j, y se define como $P_j = p_{\pi(j)}$ si el trabajo $J_{\pi(j)}$ se termina *después* de transcurrido el plazo $d_{\pi(j)}$, y $P_j = 0$ en caso contrario. El castigo total para un calendario dado es

$$P_\pi = \sum_{j=1}^{n} P_j.$$

Problema de optimización: Determinar el castigo más bajo posible (y hallar un calendario óptimo, es decir, uno que reduzca al mínimo el castigo total).

Problema de decisión: Dado, además de las entradas descritas, un entero no negativo k, ¿existe un calendario con $P_\pi \leq k$? ■

Problema 13.3 Llenado de cajones

Supóngase que tenemos un número ilimitado de cajones, cada uno con capacidad unitaria, y n objetos de tamaño $s_1, \ldots, s_n$, donde $0 < s_i \leq 1$ (los s_i son números racionales).

Problema de optimización: Determinar el número más pequeño de cajones en los cuales se pueden empacar los objetos (y hallar un empacado óptimo).

Problema de decisión: Dado, además de las entradas descritas, un entero k, ¿los objetos caben en k cajones?

Las aplicaciones del llenado de cajones incluyen el empacado de datos en memorias de computadora (por ejemplo, archivos en pistas de discos, segmentos de programa en páginas de memoria y campos de unos cuantos bits cada uno en palabras de memoria) y el surtido de pedidos de un producto (por ejemplo, tela o madera) que se cortan de piezas más grandes, de tamaño estándar. ■

Problema 13.4 Mochila

Supóngase que tenemos una mochila con capacidad C (un entero positivo) y n objetos de tamaño $s_1, \ldots, s_n$, así como "utilidades" $p_1, \ldots, p_n$ (donde $s_1, \ldots, s_n$ y $p_1, \ldots, p_n$ son enteros positivos).

Problema de optimización: Hallar la utilidad total más grande de cualquier subconjunto de los objetos que quepa en la mochila (y hallar un subconjunto que produzca la utilidad máxima).

Problema de decisión: Dado k, ¿existe un subconjunto de los objetos que quepa en la mochila y tenga una utilidad total de por lo menos k?

El problema de la mochila tiene diversas aplicaciones en planificación económica y en problemas de carga o empaque. Por ejemplo, podría describir un problema de toma de decisiones de inversión en el que el "tamaño" de una inversión es la cantidad de dinero requerida, C es el capi-

tal total que se puede invertir y la "utilidad" de una inversión es el rendimiento esperado. Sea una aplicación de una versión más complicada del problema, los objetos son tareas o experimentos que diversas organizaciones quieren que se efectúen durante un vuelo espacial. Además de su tamaño (el volumen del equipo necesario), cada tarea podría tener necesidades de energía y necesidades de tiempo de la tripulación. Tanto el espacio como la energía y el tiempo disponibles durante el vuelo son limitados. Cada tarea tiene cierto valor, o utilidad. ¿Qué subconjunto factible de las tareas tiene el valor total más grande? ■

Cabe señalar que los primeros tres problemas descritos son de minimización, mientras que el de la mochila es de maximización.

El problema que sigue es una versión más sencilla del problema de la mochila.

Problema 13.5 Sumatoria de subconjunto

La entrada es un entero positivo C y n objetos cuyos tamaños son enteros positivos $s_1, \ldots, s_n$.

Problema de optimización: Entre los subconjuntos de los objetos cuya sumatoria no es mayor que C, ¿cuál es la sumatoria de subconjunto más grande?

Problema de decisión: ¿Existe un subconjunto de los objetos cuyos tamaños sumen exactamente C? ■

Problema 13.6 Satisfactibilidad

Una variable proposicional (o booleana) es una a la que se puede asignar el valor *verdadero* o el valor *falso*. Si v es una variable proposicional, entonces $\overline{v}$, la negación de v, tiene el valor *verdadero* si y sólo si v tiene el valor *falso*. Una *literal* es una variable proposicional o la negación de una variable proposicional. Una fórmula proposicional se define inductivamente como una expresión que es una variable proposicional, una constante proposicional (es decir, *verdadero* o *falso*) o una expresión que consiste en un operador booleano y sus operandos, que son fórmulas proposicionales. Hay varias formas de representar una fórmula proposicional, que incluyen la notación funcional (por ejemplo, $and(x, y)$), la notación de operadores (por ejemplo, $(x \wedge y)$) y los árboles de expresión en los que cada nodo interno es un operador booleano y cada hoja es una variable proposicional o una de las constantes *verdadero* o *falso*. Si asignamos valores de verdad a las variables, la fórmula tendrá un valor de verdad que se obtiene aplicando las reglas de los operadores.

Resulta muy útil cierta forma regular para las fórmulas proposicionales, llamada *forma normal conjuntiva*. Una *cláusula* es una sucesión de literales separadas por el operador *or* booleano ($\vee$). Una fórmula proposicional está en *forma normal conjuntiva* (*CNF*, por sus siglas en inglés) si consiste en una sucesión de cláusulas separadas por el operador booleano *and* ($\wedge$). Un ejemplo de fórmula proposicional en CNF es

$$(p \vee q \vee s) \wedge (\overline{q} \vee r) \wedge (\overline{p} \vee r) \wedge (\overline{r} \vee s) \wedge (\overline{p} \vee \overline{s} \vee \overline{q})$$

donde p, q, r y s son variables proposicionales. En todo este capítulo, "fórmula CNF" siempre se refiere a una fórmula proposicional en CNF.

Una *asignación de verdad* a un conjunto de variables proposicionales es la asignación de uno de los valores *verdadero* o *falso* a cada una de las variables proposicionales del conjunto, o sea, una función con valores booleanos sobre el conjunto. Decimos que una asignación de verdad *sa-*

tisface una fórmula si hace que el valor de toda la fórmula sea *verdadero*. Cabe señalar que una fórmula CNF se satisface si y sólo si la evaluación de cada una de las cláusulas da *verdadero*, lo cual sucede si y sólo si por lo menos una literal de la cláusula es *verdadera*.

Problema de decisión: Dada una fórmula CNF, ¿existe alguna asignación de verdad que la satisfaga?

Este problema de decisión se denomina *satisfactibilidad CNF*, o simplemente *satisfactibilidad*, y a menudo se abrevia *CNF-SAT* o *SAT*. El problema de satisfactibilidad tiene aplicaciones en la demostración automatizada de teoremas, y desempeñó un papel central en el desarrollo de las ideas de este capítulo.

La simplificación siguiente del problema de la satisfactibilidad, llamada *3-satisfactibilidad*, *3-satisfactibilidad CNF*, *3-SAT* y *3-CNF-SAT*, también es importante. (Enumeramos todos los nombres y abreviaturas porque la nomenclatura no es estándar, y el problema se menciona a menudo.)

Problema de decisión: Dada una fórmula CNF, en la que ninguna cláusula puede contener más de tres literales, ¿existe alguna asignación de verdad que la satisfaga? ■

Problema 13.7 Ciclos hamiltonianos y caminos hamiltonianos

Un *ciclo hamiltoniano* en un grafo no dirigido es un ciclo simple que pasa exactamente una vez por cada uno de los vértices. A veces se usa la palabra *circuito* en vez de *ciclo*.

Problema de decisión: ¿Un ciclo no dirigido dado tiene un ciclo hamiltoniano?

Un problema de optimización relacionado es el del vendedor viajero, o de recorrido mínimo, que describiremos después de éste.

Un *camino hamiltoniano* en un grafo no dirigido es un camino simple que pasa por cada uno de los vértices exactamente una vez.

Problema de decisión: ¿Un grafo no dirigido dado tiene un camino hamiltoniano?

Ambos problemas podrían plantearse también para grafos *dirigidos*, en cuyo caso se denominan "problema del ciclo (o camino) hamiltoniano dirigido". Una variante del problema del camino hamiltoniano incluye un vértice inicial y final específico para el camino. ■

Problema 13.8 Vendedor viajero

Este problema se conoce ampliamente como problema del vendedor viajero (*TSP*, por sus siglas en inglés) pero también se conoce como el *problema de recorrido mínimo*. El vendedor quiere reducir al mínimo el costo de viaje total (en tiempo o en distancia) requerido para visitar todas las ciudades de un territorio y volver al punto de partida. Otras aplicaciones incluyen la determinación de rutas para los camiones que recogen la basura o que entregan paquetes.

Problema de optimización: Dado un grafo ponderado completo, hallar un ciclo hamiltoniano de peso mínimo.

Problema de decisión: Dado un grafo ponderado completo y un entero k, ¿existe un ciclo hamiltoniano cuyo peso total no sea mayor que k?

La versión tradicional trata el grafo como no dirigido; es decir, los pesos son iguales en ambos sentidos. Al igual que con el problema del ciclo hamiltoniano, existe una versión dirigida. ■

La utilidad y aparente sencillez de estos problemas podría dejar intrigado al lector; le invitamos a tratar de idear algoritmos para algunos de ellos antes de proceder.

13.2.3 La clase $\mathcal{P}$

Ninguno de los algoritmos que se conocen para los problemas que acabamos de describir tiene garantía de ejecutarse en un tiempo razonable. No definiremos "razonable" de forma rigurosa, pero sí definiremos una clase $\mathcal{P}$ de problemas que *incluye* a los que tienen algoritmos razonables.

Definición 13.2 Polinómicamente acotado

Decimos que un algoritmo está *polinómicamente acotado* si su complejidad de peor caso está acotada por una función polinómica del tamaño de las entradas (es decir, si existe un polinomio p tal que, para toda entrada de tamaño n, el algoritmo termine después de cuando más $p(n)$ pasos).

Decimos que un problema está polinómicamente acotado si existe un algoritmo polinómicamente acotado para resolverlo. ■

Todos los problemas y algoritmos que estudiamos en los capítulos 1 a 12 están polinómicamente acotados, con la excepción de uno que otro ejercicio.

Definición 13.3 La clase $\mathcal{P}$

$\mathcal{P}$ es la clase de problemas de decisión que están polinómicamente acotados. ■

$\mathcal{P}$ sólo está definida para problemas de decisión, pero en general no nos equivocaremos mucho si consideramos que los tipos de problemas que estudiamos en los capítulos anteriores del libro están en $\mathcal{P}$.

Podría parecer un tanto extravagante utilizar la existencia de una cota de tiempo polinómica como criterio para definir la clase de problemas más o menos razonables: los polinomios pueden ser muy grandes. No obstante, hay varias razones de peso para hacerlo.

En primer lugar, si bien no es verdad que todos los problemas en $\mathcal{P}$ tengan un algoritmo de eficiencia aceptable, sí podemos asegurar que, si un problema *no* está en $\mathcal{P}$, será extremadamente costoso y probablemente imposible de resolver en la práctica. Es probable que ninguno de los problemas que describimos al principio de esta sección esté en $\mathcal{P}$; no existen algoritmos para resolverlos que se sepa estén polinómicamente acotados, y la mayoría de los investigadores en el campo piensan que no existen tales algoritmos. Así pues, si bien es factible que la definición de $\mathcal{P}$ sea demasiado amplia para servir como criterio en el caso de problemas con necesidades de tiempo verdaderamente razonables, el hecho de estar o no en $\mathcal{P}$ sí es un criterio útil en el caso de problemas renuentes.

Un segundo motivo para usar una cota polinómica para definir $\mathcal{P}$ es que los polinomios tienen bonitas propiedades de "cierre". Podríamos obtener un algoritmo para un problema complejo combinando varios algoritmos para problemas más sencillos. Algunos de los algoritmos más

simples podrían trabajar con las salidas o los resultados intermedios de otros. La complejidad del algoritmo compuesto podría estar acotada por la suma, multiplicación y composición de las complejidades de sus algoritmos componentes. Puesto que los polinomios están cerrados bajo estas operaciones, cualquier algoritmo que se construya de diversas formas naturales a partir de varios algoritmos polinómicamente acotados también estará polinómicamente acotado. Ninguna clase más pequeña de funciones que sean cotas de complejidad útiles tiene estas propiedades de cierre.

Un tercer motivo para emplear una cota polinómica es que hace a $\mathcal{P}$ independiente del modelo de cómputo formal específico que se use. Se usan varios modelos formales (definiciones formales de algoritmos) para demostrar teoremas rigurosos acerca de la complejidad de los algoritmos y problemas. Los modelos difieren en cuanto al tipo de operaciones permitidas, los recursos de memoria disponibles y los costos asignados a diferentes operaciones. Un problema que requiere $\Theta(f(n))$ pasos con un modelo podría requerir más de $\Theta(f(n))$ pasos con otro, pero en el caso de prácticamente todos los modelos realistas, si un problema está acotado polinómicamente con uno, estará acotado polinómicamente con los demás.

13.2.4 La clase $\mathcal{NP}$

Muchos problemas de decisión (incluidos todos los que hemos puesto de muestra) se plantean como preguntas de existencia: ¿Existe un k-coloreado del grafo G? ¿Existe una asignación de verdad que haga verdadera una fórmula CNF dada? Para una entrada dada, una "solución" es un objeto (por ejemplo, un coloreado de grafo o una asignación de verdad) que satisface los criterios del problema y por tanto justifica una respuesta de *sí* (por ejemplo, el coloreado de grafo emplea cuando más k colores; la asignación de verdad hace que la fórmula CNF sea verdadera). Una "solución propuesta" no es más que un objeto del tipo apropiado: podría satisfacer o no los criterios. A veces se usa el término *certificado* para referirse a una solución propuesta. En términos informales, $\mathcal{NP}$ es la clase de problemas de decisión para los que una solución propuesta dada con una entrada dada se puede verificar rápidamente (en tiempo polinómico) para determinar si realmente es una solución (es decir, si satisface los requisitos del problema). Más formalmente, las entradas de un problema y las soluciones propuestas se deberán describir con cadenas de símbolos de algún conjunto finito, por ejemplo, el conjunto de caracteres del teclado de una terminal de computadora. Necesitamos algunas convenciones para describir grafos, conjuntos, funciones, etc., que usan tales símbolos. El conjunto de convenciones que se adopta para un problema dado es la *codificación* de ese problema. El tamaño de una cadena es el número de caracteres que contiene. Algunas cadenas de símbolos del conjunto escogido no son codificaciones válidas de objetos pertinentes al problema en cuestión; sólo son basura. En términos formales, una entrada de un problema y una solución propuesta para ese ejemplar del problema pueden ser cualquier cadena formada a partir del conjunto de caracteres. La verificación de una solución propuesta incluye verificar que la cadena tenga sentido (es decir, que tenga la sintaxis correcta) como descripción del tipo de objeto requerido, además de verificar que satisfaga los criterios del problema. Por tanto, cualquier cadena de caracteres puede verse como un certificado para un ejemplar de un problema.

Podría haber problemas de decisión para los que no haya una interpretación natural de las "soluciones" o las "soluciones propuestas". En lo abstracto, un problema de decisión no es más que una función que relaciona un conjunto de cadenas de entrada con el conjunto {*sí*, *no*}. Una definición formal de $\mathcal{NP}$ considera todos los problemas de decisión. La definición emplea algorit-

mos no deterministas, que definiremos a continuación. Aunque tales algoritmos no son realistas ni útiles en la práctica, sirven para clasificar los problemas.

Definición 13.4 Algoritmo no determinista

Un *algoritmo no determinista* tiene dos fases y un paso de salida:

1. La fase de "conjetura" no determinista. Se escribe alguna cadena totalmente arbitraria de caracteres, *s*, principiando en algún lugar designado de la memoria. Cada vez que se ejecuta el algoritmo, la cadena que se escribe podría ser distinta. (Esta cadena es el certificado; podría considerarse como una conjetura de la solución del problema, por lo que podríamos llamar a ésta la fase de conjetura, pero *s* bien podría ser sólo basura.)
2. La fase de "verificación" determinista. Comienza a ejecutarse una subrutina determinista (o sea, ordinaria). Además de las entradas del problema de decisión, la subrutina podría usar *s*, o podría hacer caso omiso de *s*. Tarde o temprano devolverá un valor de *verdadero* o *falso*, o podría entrar en un ciclo infinito y nunca parar. (Podemos pensar en la fase de verificación como un examen de *s* encaminado a determinar si es una solución para las entradas del problema de decisión, es decir, si justifica una respuesta de *sí* para las entradas del problema de decisión.)
3. El paso de salida. Si la fase de verificación devolvió *verdadero*, el algoritmo produce *sí*. De lo contrario, no se produce ninguna salida. ■

El número de pasos que se ejecutan durante una ejecución de un algoritmo no determinista se define como la suma de los pasos de las dos fases; es decir, el número de pasos que se ejecutan para escribir *s* (que es simplemente el número de caracteres de *s*) más el número de pasos que ejecuta la segunda fase determinista.

También podemos describir un algoritmo no determinista con una estructura de subrutina explícita. Supóngase que `genCertif` genera un certificado arbitrario.

```
void aNoDet(String entrada)
    String s = genCertif();
    boolean revisOK = verifA(entrada, s);
    if (revisOK)
       Enviar a la salida "sí".
    return;
```

Normalmente, los algoritmos terminan con todas las entradas y cada vez que ejecutamos un algoritmo con la misma entrada obtenemos la misma salida. Esto no sucede con los algoritmos no deterministas; con una entrada específica *x*, la salida (o ausencia de salida) de una ejecución podría diferir de la de otra porque podría depender de *s*. Entonces, ¿qué "respuesta" calcula un algoritmo no determinista, digamos **A**, para un problema de decisión dado con la entrada *x*? La respuesta de **A** con *x* se define como *sí* si y sólo si existe *alguna* ejecución de **A** que produzca *sí* como salida. La respuesta es *no* si, para toda *s*, no se produce salida. Utilizando nuestro concepto informal de *s* como una solución propuesta, la respuesta de **A** con *x* es *sí* si y sólo si existe alguna solución propuesta que "funcione".

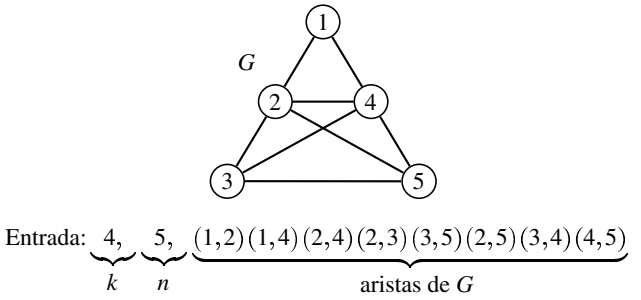


Figura 13.1 Entrada para un coloreado de grafos no determinista (ejemplo 13.1)

Ejemplo 13.1 Coloreado no determinista de grafos

Supóngase que el problema consiste en determinar si un grafo no dirigido G es k-coloreable. La primera fase de un algoritmo no determinista escribirá alguna cadena s, que la segunda fase puede interpretar como un coloreado propuesto. La cadena s se puede interpretar como una lista de enteros $c_1, c_2, \ldots, c_q$ para alguna q que depende de la longitud de s. La segunda fase del algoritmo puede interpretar estos enteros como colores que se asignarán a los vértices: asignar c_i a v_i. Para verificar si el coloreado es válido, la segunda fase ejecuta estos pasos:

1. Verificar que la lista incluya n colores (es decir, que $q = n$).
2. Verificar que todo c_i esté dentro del intervalo $1, \ldots, k$.
3. Explorar la lista de aristas del grafo (o una matriz de adyacencia) y, para cada arista v_iv_j, verificar que $c_i \neq c_j$; es decir, que los dos vértices que inciden en una arista tienen colores distintos.

Si se satisfacen todas estas pruebas, el verificador devuelve *verdadero* y el algoritmo produce *sí*. Si s no satisface todos los requisitos, el verificador podría devolver *falso* o entrar en un ciclo infinito, y el algoritmo no producirá salida en esta ejecución.

Por ejemplo, sea el ejemplar de entrada el grafo G de la figura 13.1 y sea $k = 4$, así que la pregunta en este caso es: "¿Se puede 4-colorear G?" Para facilitar la lectura, denotaremos los colores con letras A (azul), R (rojo), V (verde), Y (amarillo) y N (naranja), en lugar de con los enteros $1, \ldots, 5$. He aquí una lista de unas pocas cadenas certificado s posibles y los valores que el verificador devuelve.

s	Salida	Motivo
RVRAV	*falso*	v_2 y v_5, ambos verdes, están adyacentes
RVRA	*falso*	No están coloreados todos los vértices
RAYVN	*falso*	Se usaron demasiados colores
RVRAY	*verdadero*	4-coloreado válido
R%*,V@	*falso*	Sintaxis errónea

Puesto que hay (al menos) un posible cálculo del verificador que devuelve *verdadero*, la respuesta del algoritmo no determinista con la entrada $(G, 4)$ es *sí*. ■

Decimos que un algoritmo no determinista está polinómicamente acotado si existe un polinomio (fijo) p tal que para toda entrada de tamaño n para la cual la respuesta es *sí* existe alguna ejecución del algoritmo que produce una salida *sí* en no más de $p(n)$ pasos.

Definición 13.5 La clase $\mathcal{NP}$

$\mathcal{NP}$ es la clase de problemas de decisión para la cual existe un algoritmo no determinista polinómicamente acotado. (El nombre $\mathcal{NP}$ proviene de "*N*o determinista *P*olinómicamente acotado".) ■

Teorema 13.1 Los problemas 13.1 a 13.8: coloreado de grafos, ciclo hamiltoniano, camino hamiltoniano, calendarización de trabajos con castigos, llenado de cajones, sumatoria de subconjunto, mochila, satisfactibilidad y vendedor viajero están todos en $\mathcal{NP}$.

Demostración Las demostraciones son directas y se dejan como ejercicios. Por ejemplo, el trabajo requerido para verificar un posible coloreado de grafo, que describimos antes, se puede efectuar fácilmente en tiempo polinómico. □

Teorema 13.2 $\mathcal{P} \subseteq \mathcal{NP}$.

Demostración Un algoritmo ordinario (determinista) para resolver un problema de decisión es, con una modificación menor, un caso especial de un algoritmo no determinista. Si **A** es un algoritmo determinista para resolver un problema de decisión, basta con hacer que **A** sea la segunda fase de un algoritmo no determinista, pero modificándolo de modo que, en todos los casos en que produciría la salida *sí*, devuelva *verdadero*, y en todos los casos en que produciría la salida *no*, devuelva *falso*. **A** simplemente hará caso omiso de lo que haya escrito la primera fase y procederá a efectuar su cálculo acostumbrado. Un algoritmo no determinista puede ejecutar cero pasos en la primera fase (escribiendo la cadena nula) de modo que, si **A** se ejecuta en tiempo polinómico, el algoritmo no determinista que tiene el **A** modificado como segunda fase también se ejecutará en tiempo polinómico. Producirá *sí* si **A** lo habría hecho, y no producirá nada en caso contrario. ■

La pregunta importante es, ¿$\mathcal{P} = \mathcal{NP}$ o es $\mathcal{P}$ un subconjunto propio de $\mathcal{NP}$? Dicho de otro modo, ¿es el no determinismo más potente que el determinismo en el sentido de que algunos problemas se pueden resolver en tiempo polinómico con un "conjeturador" no determinista pero no se pueden resolver en tiempo polinómico con un algoritmo ordinario? Si un problema está en $\mathcal{NP}$, con una cota de tiempo polinómico, digamos p, podremos dar (de forma determinista) la respuesta correcta (*sí* o *no*) si verificamos todas las cadenas cuya longitud no sea mayor que $p(n)$ (es decir, ejecutamos la segunda fase del algoritmo no determinista con cada una de las cadenas posibles, una por una). El número de pasos requeridos para verificar cada cadena es cuando más $p(n)$. El problema es que hay demasiadas cadenas que verificar. Si nuestro conjunto de caracteres contiene c caracteres, habrá $c^{p(n)}$ cadenas de longitud $p(n)$. El número de cadenas es exponencial, no polinómico, en n. Claro que hay otra forma de resolver problemas: usar algunas de las propiedades de los objetos en cuestión y un poco de ingenio para idear un algoritmo que no tenga que examinar todas las posibles soluciones. Al ordenar, por ejemplo, no verificamos cada una de las $n!$ permutaciones de las n claves dadas para ver cuál coloca las claves en orden. La dificultad, en el caso

de los problemas que tratamos en este capítulo, estriba en que tal enfoque no ha producido algoritmos eficientes; todos los algoritmos conocidos examinan todas las posibilidades o bien, si se valen de ardides para reducir el trabajo, tales ardides no son lo bastante buenos como para dar pie a algoritmos polinómicamente acotados.

Se cree que $\mathcal{NP}$ es un conjunto mucho más grande que $\mathcal{P}$, pero no existe un solo problema en $\mathcal{NP}$ para el cual se haya demostrado que el problema no está en $\mathcal{P}$. No se conocen algoritmos polinómicamente acotados para muchos problemas que están en $\mathcal{NP}$ (incluidos todos los problemas de muestra de la sección 13.2.2), pero no se han demostrado cotas inferiores mayores que las polinómicas para esos problemas. Por tanto, la pregunta que hicimos antes, ¿es $\mathcal{P} = \mathcal{NP}$? sigue pendiente.

13.2.5 El tamaño de las entradas

Consideremos el problema siguiente.

Problema 13.9

Dado un entero positivo n, ¿existen enteros j, $k > 1$ tales que $n = jk$? (es decir, ¿es n no primo?).
■

¿Este problema está en $\mathcal{P}$? Consideremos el algoritmo siguiente, que busca un factor de n.

```
factor = 0
for (j = 2; j < n; j++)
    if ((n mod j) == 0)
        factor = j;
        break;
return factor;
```

El cuerpo del ciclo se ejecuta menos de n veces, y es indudable que (n mod j) se puede evaluar en $O(\log^2(n))$, así que el tiempo de ejecución del algoritmo está holgadamente en $O(n^2)$. Sin embargo, *no* se sabe que el problema de determinar si un entero es primo o es factorizable esté en $\mathcal{P}$ y, de hecho, la dificultad para hallar factores de enteros grandes es la base de varios algoritmos de cifrado precisamente porque se le considera un problema difícil. ¿Cómo resolvemos esta aparente paradoja?

La entrada del algoritmo para probar primos es el entero n pero, ¿qué tamaño tiene la entrada? Hasta ahora, hemos utilizado cualquier medida cómoda y razonable del tamaño de las entradas; no era importante contar caracteres o bits individuales. Si nuestra medida del tamaño de una entrada podría marcar la diferencia entre si un algoritmo es polinómico o es exponencial, hay que tener más cuidado. El tamaño de una entrada es el número de caracteres que se requieren para escribirla. Si $n = 150$, por ejemplo, escribimos tres dígitos, no 150 dígitos. Así pues, un entero n escrito en notación decimal tiene un tamaño aproximado de $\log_{10} n$. Si optamos por considerar la representación interna en una computadora, donde los enteros se representan en binario, el tamaño de n será aproximadamente lg n. Estas representaciones difieren en un factor constante; es decir, $\log_2 n = \log_2 10 \log_{10} n$, así que no es crucial cuál representación usemos. Lo que sí es importante es que, si el tamaño s de las entradas es $\log_{10} n$ y el tiempo de ejecución de un algoritmo es n, el tiempo de ejecución del algoritmo será una función exponencial del tamaño de las

entradas ($n = 10^s$). Por tanto, el algoritmo anterior para determinar si n es primo no está en $\mathcal{P}$. No se conocen aún algoritmos para probar primos en tiempo polinómico. Sin embargo, la pregunta: "¿El entero n es primo?" está en $\mathcal{NP}$ (véase el ejercicio 13.4).

En los problemas que estudiamos en capítulos anteriores, la variable que usábamos para describir el tamaño de las entradas correspondía (aproximadamente) a la cantidad de datos contenida en ellas. Por ejemplo, usamos n como tamaño de la entrada al ordenar una lista de n claves. Cada una de las claves se representaría, digamos, en binario, pero dado que hay n claves, habrá por lo menos n símbolos en la entrada. Por tanto, si la complejidad de un algoritmo está acotada por un polinomio en n, estará acotada por un polinomio en el tamaño exacto de la entrada.

Asimismo, usamos $(m + n)$ como el tamaño de las entradas de grafos, pero es necesario enumerar explícitamente todas las aristas, lo cual requiere al menos m símbolos en la entrada. Aunque no es necesario enumerar los n vértices en la entrada, en todos los problemas de interés todo vértice incide en alguna arista, así que $(n + m)$ es cuando más tres veces el número de símbolos que hay en la entrada. Una vez más, si la complejidad de un algoritmo está acotada por un polinomio en $(n + m)$, estará acotada por un polinomio en el tamaño exacto de la entrada.

Si dos medidas del tamaño de las entradas están acotadas cada una por una función polinómica de la otra, determinar si el problema está en $\mathcal{P}$ no dependerá de la medida específica que se use. En el ejemplo de ordenamiento, si una medida es el número de claves, n, y la otra es n lg(clave máxima), que cuenta bits individuales, tendremos $n \in O(n \log(\text{clave máxima}))$ y n lg(clave máxima) $\in O(n^2)$. Por tanto, cada una de las medidas no está a más de una función polinómica de la otra.

Por ello, normalmente no tenemos que especificar con toda precisión el tamaño de las entradas. No obstante, debemos tener cuidado en los casos en que el tiempo de ejecución de un algoritmo se expresa como función polinómica de uno de los *valores* de entrada, como sucede con el problema de la prueba de primos.

Unos cuantos de los problemas de muestra que describimos antes tienen soluciones de programación dinámica que a primera vista parecen estar polinómicamente acotados pero, al igual que el programa de prueba de primos, no lo están. Por ejemplo, recordemos el problema de la sumatoria de subconjunto: ¿Existe un subconjunto de los n objetos con tamaños $s_1, s_2, \ldots, s_n$ cuya suma sea exactamente C? Si utilizamos las técnicas del capítulo 10, podremos resolver este problema con una tabla de n por C y sólo requeriremos unas cuantas operaciones para calcular cada elemento de la tabla (véase el ejercicio 13.5a). Existen soluciones de programación dinámica similares para diversas versiones del problema de la mochila.

La solución de programación dinámica para el problema de la suma de subconjunto se ejecuta en tiempo $\Theta(nC)$. Puesto que hay n objetos en la entrada, el término n no representa ningún problema, pero el valor del número C es exponencialmente mayor (en general) que la entrada, porque el dato C en la entrada se representaría con lg C bits. Por ello, la solución de programación dinámica no es un algoritmo polinómicamente acotado. Desde luego, si C no es demasiado grande, el algoritmo podría ser útil en la práctica.

13.3 Problemas $\mathcal{NP}$-completos

Usamos el término *$\mathcal{NP}$-completo* para describir los problemas de decisión más difíciles en $\mathcal{NP}$ en el sentido de que, si existiera un algoritmo polinómicamente acotado para un problema $\mathcal{NP}$-completo, existiría un algoritmo polinómicamente acotado para todos los problemas en $\mathcal{NP}$.

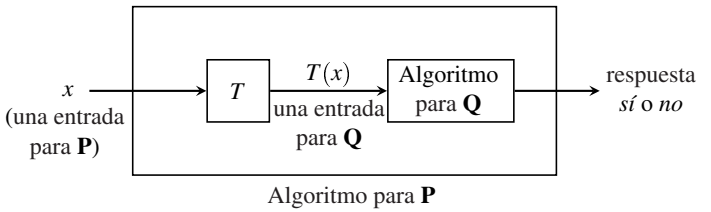


Figura 13.2 Reducción de un problema **P** a un problema **Q**: la respuesta del problema **Q** con $T(x)$ debe ser la misma que la respuesta de **P** con x.

Algunos de los problemas de muestra que describimos en la sección 13.2.2 podrían parecer más fáciles que otros y, de hecho, las complejidades de peor caso de los algoritmos que se han ideado para resolverlos sí difieren (son funciones de crecimiento rápido como $2^{\sqrt{n}}$, 2^n, $(n/2)^{n/2}$, $n!$, etc.) pero, lo cual es sorprendente, todos son equivalentes en el sentido de que, si cualquiera está en $\mathcal{P}$, todos lo están. Todos son $\mathcal{NP}$-completos.

13.3.1 Reducciones polinómicas

La definición formal de "$\mathcal{NP}$-completo" emplea reducciones, o transformaciones, de un problema a otro. Supóngase que nos interesa resolver un problema **P** y ya tenemos un algoritmo para otro problema **Q**. Supóngase que también tenemos una función T que toma una entrada x para **P** y produce $T(x)$, una entrada para **Q** tal que la respuesta correcta para **P** con x es *sí* si y sólo si la respuesta correcta para **Q** con $T(x)$ es *sí*. Entonces, componiendo T y el algoritmo para resolver **Q**, tendremos un algoritmo para resolver **P**. Véase la figura 13.2.

Ejemplo 13.2 Una reducción simple

Sea el problema **P**: Dada una sucesión de valores booleanos, ¿al menos uno de ellos tiene el valor *verdadero*? (En otras palabras, se trata del cálculo del *or* booleano de n vías cuando la cadena tiene n valores, planteado como problema de decisión.)

Sea **Q**: Dada una sucesión de enteros, ¿el máximo de los enteros es positivo?

Definamos la transformación T así:

$$T(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$$

donde $y_i = 1$ si $x_i = verdadero$, y $y_i = 0$ si $x_i = falso$.

Es evidente que, si aplicamos a $y_1, y_2, \ldots, y_n$ un algoritmo para resolver **Q**, resolveremos **P** para $x_1, x_2, \ldots, x_n$. ■

Definición 13.6 Reducción polinómica y reducibilidad

Sea T una función que establece una correspondencia entre el conjunto de entrada de un problema de decisión **P** y el conjunto de entrada de un problema de decisión **Q**. T es una *reducción polinómica* (también llamada *transformación polinómica*) de **P** a **Q** si se cumplen todas las condiciones siguientes:

1. T se puede calcular en tiempo polinómicamente acotado.

2. Para toda cadena x, si x es una entrada de *sí* para **P**, $T(x)$ es una entrada de *sí* para **Q**.

3. Para toda cadena x, si x es una entrada de *no* para **P**, $T(x)$ es una entrada de *no* para **Q**.

Por lo regular es más fácil demostrar la contrapositiva (sección 1.3.3) de la parte 3:

3′. Para toda x, si $T(x)$ es una entrada de *sí* para **Q**, x es una entrada de *sí* para **P**.

El problema **P** es *polinómicamente reducible* (o *polinómicamente transformable*) a **Q** si existe una transformación polinómica de **P** a **Q**. (En este capítulo por lo regular diremos sólo que **P** *se puede reducir* a **Q**; la cota polinómica se sobreentiende.) Usamos la notación

$$\mathbf{P} \leq_P \mathbf{Q}$$

para indicar que **P** se puede reducir a **Q**. ■

Cabe señalar que las partes 2 y 3 (o 3′) de la definición de *reducción* se combinan para decir que $T(x)$ tiene la *misma* respuesta para el problema **Q** que x tiene para **P**, para toda x.

El meollo de la reducibilidad de **P** a **Q** es que **Q** es por lo menos tan "difícil" de resolver como **P**. El teorema que sigue plantea esto con mayor precisión.

Teorema 13.3 Si $\mathbf{P} \leq_P \mathbf{Q}$ y **Q** está en $\mathcal{P}$, entonces **P** está en $\mathcal{P}$.

Demostración Sea p una cota polinómica para el cálculo de T, y sea q una cota polinómica para un algoritmo que resuelve **Q**. Sea x una entrada de tamaño n para **P**. Entonces el tamaño de $T(x)$ no es mayor que $p(n)$ (puesto que, en el peor de los casos, un programa para efectuar T escribe un símbolo en cada paso). Si se proporciona $T(x)$ al algoritmo para resolver **Q**, dicho algoritmo ejecuta cuando más $q(p(n))$ pasos. Así pues, la cantidad total de trabajo efectuada para transformar x en $T(x)$ y luego usar el algoritmo que resuelve **Q** para obtener la respuesta correcta de **P** con x es $p(n) + q(p(n))$, un polinomio en n. □

Ya podemos dar la definición formal de $\mathcal{NP}$-completo.

Definición 13.7 $\mathcal{NP}$-difícil y $\mathcal{NP}$-completo

Un problema **Q** es $\mathcal{NP}$-*difícil* si todo problema **P** en $\mathcal{NP}$ se puede reducir a **Q**; es decir, $\mathbf{P} \leq_P \mathbf{Q}$. Un problema **Q** es $\mathcal{NP}$-*completo* si está en $\mathcal{NP}$ y es $\mathcal{NP}$-*difícil*. ■

Es importante darse cuenta de que "$\mathcal{NP}$-difícil" *no* significa "en $\mathcal{NP}$ y difícil"; significa "al menos tan difícil como cualquier problema en $\mathcal{NP}$". Así, un problema puede ser $\mathcal{NP}$-difícil y *no* estar en $\mathcal{NP}$.

Ser $\mathcal{NP}$-difícil constituye una cota inferior para el problema. Estar en $\mathcal{NP}$ constituye una cota superior. Así pues, la clase de los problemas $\mathcal{NP}$-completos está acotada tanto por abajo como por arriba, aunque el estado actual del conocimiento humano no permite definir de forma nítida ninguna de esas fronteras. El teorema que sigue es consecuencia directa de la definición y del teorema 13.3.

Teorema 13.4 Si cualquier problema $\mathcal{NP}$-completo está en $\mathcal{P}$, entonces $\mathcal{P} = \mathcal{NP}$. □

Este teorema indica, por una parte, lo valioso que sería hallar un algoritmo polinómicamente acotado para cualquier problema $\mathcal{NP}$-completo y, por la otra, lo poco probable que es la existen-

cia de tal algoritmo, en vista de la gran cantidad de problemas en $\mathcal{NP}$ para los que se han buscado infructuosamente algoritmos polinómicamente acotados.

Aunque hemos visto que muchos problemas están en $\mathcal{NP}$, no es de ninguna manera evidente que cualquiera de ellos sea $\mathcal{NP}$-completo. Después de todo, para demostrar que algún problema **Q** es $\mathcal{NP}$-difícil, lo cual es la segunda mitad del requisito para ser $\mathcal{NP}$-completo, es necesario demostrar que *todos* los problemas en $\mathcal{NP}$, incluso los que no conocemos, se pueden reducir al problema específico **Q**. ¿Cómo podríamos siquiera comenzar a atacar tal labor? La primera demostración de que cierto problema efectivamente es $\mathcal{NP}$-completo es uno de los mayores logros de la teoría de la computación y de las matemáticas.

Teorema 13.5 (Teorema de Cook) El problema de la satisfactibilidad es $\mathcal{NP}$-completo. □

La demostración de este teorema, y de otros que presentamos aquí sin demostrarlos, se puede hallar en las fuentes que se dan en Notas y referencias al final del capítulo. Aquí sólo bosquejaremos la idea. La demostración debe mostrar que cualquier problema **P** en $\mathcal{NP}$ se puede reducir a la satisfactibilidad. La demostración de Steven Cook da un algoritmo para construir una fórmula CNF para una entrada x de **P** tal que la fórmula, en términos informales, describe el cálculo efectuado por un algoritmo no determinista para resolver **P** al actuar sobre x. La fórmula CNF, que es muy larga pero se construye en tiempo acotado por una función polinómica de la longitud de x, se podrá satisfacer si y sólo si el cálculo produce la respuesta *sí* con algún certificado.

La reducción polinómica es una relación transitiva (véase el ejercicio 13.6). Por tanto, si la satisfactibilidad se puede reducir a algún problema **Q**, **Q** también es $\mathcal{NP}$-difícil. Si **Q** también está en $\mathcal{NP}$ (lo cual por lo regular se puede demostrar fácilmente), entonces **Q** es $\mathcal{NP}$-completo. Así, la reducción sirve para demostrar que otros problemas son $\mathcal{NP}$-completos sin tener que repetir todo el trabajo del teorema de Cook. Por ejemplo, la satisfactibilidad se puede reducir a 3-satisfactibilidad (véase el ejercicio 13.7).

La satisfactibilidad (y la 3-satisfactibilidad) son problemas lógicos, y no tienen una relación obvia con los demás problemas que describimos en la sección 13.2.2 ni con los muchos otros problemas de optimización que no parecen tener una solución eficiente, algunos en el campo de los grafos y otros en el de los compiladores y los sistemas operativos. Si los únicos problemas $\mathcal{NP}$-completos fueran parecidos al de la satisfactibilidad, la calidad de ser $\mathcal{NP}$-completo no habría pasado de ser una curiosidad interesante.

El segundo trabajo crucial en el campo fue el escrito por Richard Karp, quien demostró que los planteamientos como problemas de decisión de un gran número de problemas de optimización, incluidos varios de los problemas muestra que describimos antes, también son $\mathcal{NP}$-completos. Se requirieron reducciones muy ingeniosas para demostrar que los problemas se podrían reducir a otros al parecer muy distintos. Por ejemplo, Karp mostró que la 3-satisfactibilidad se podía reducir (mediante una cadena de reducciones, en algunos casos) a problemas de grafos que al parecer no tenían ninguna relación, como el del ciclo hamiltoniano y el de coloreado de grafos. Esto fue el inicio de una avalancha. Pronto se demostró que muchos problemas para los cuales se estaban buscando infructuosamente algoritmos polinómicamente acotados eran $\mathcal{NP}$-completos. De hecho, la lista de problemas $\mathcal{NP}$-completos llegó a los centenares en los años setenta.

Teorema 13.6 Los problemas de coloreado de grafos, ciclo hamiltoniano, camino hamiltoniano, calendarización de trabajos con castigos, llenado de cajones, la sumatoria de subconjuntos, la mochila y el vendedor viajero son todos $\mathcal{NP}$-completos. □

Como ya dijimos, para demostrar que un problema $\mathbf{Q} \in \mathcal{NP}$ es $\mathcal{NP}$-completo basta con demostrar que algún otro problema $\mathcal{NP}$-completo se puede reducir polinómicamente a **Q**, pues la relación de reducibilidad es transitiva. Por ello, las diversas partes del teorema 13.6 se demuestran estableciendo cadenas de transformaciones que parten del problema de la satisfactibilidad. Presentaremos unas cuantas como ejemplo.

Muchos estudiantes se confunden en cuanto a la dirección de la reducción que se requiere para demostrar que un problema es $\mathcal{NP}$-completo, por lo que cabe hacer hincapié en que, para demostrar que un problema **Q** es $\mathcal{NP}$-completo, se escoge algún problema $\mathcal{NP}$-completo conocido **P** y se reduce **P** a **Q**, no al revés. La lógica es la siguiente:

1. Puesto que **p** es $\mathcal{NP}$-completo, todos los problemas $\mathbf{R} \in \mathcal{NP}$ se pueden reducir a **P**; es decir, $\mathbf{R} \leq_P \mathbf{P}$.
2. Demostrar que $\mathbf{P} \leq_P \mathbf{Q}$.
3. Entonces todos los problemas $\mathbf{R} \in \mathcal{NP}$ satisfacen $\mathbf{R} \leq_P \mathbf{Q}$, por la transitividad de las reducciones.
4. Por tanto, **Q** es $\mathcal{NP}$-completo.

Teorema 13.7 El problema del ciclo hamiltoniano dirigido se puede reducir al problema del ciclo hamiltoniano no dirigido. (Por tanto, si sabemos que el problema del ciclo hamiltoniano dirigido es $\mathcal{NP}$-completo, podremos concluir que el problema del ciclo hamiltoniano no dirigido también es $\mathcal{NP}$-completo.)

Demostración Sea $G = (V, E)$ un grafo dirigido con n vértices. G se transforma en el grafo no dirigido $G' = (V', E')$ donde, para cada vértice $v \in V$, el conjunto de vértices transformados V' contiene tres vértices llamados v^1, v^2 y v^3. También, para cada $v \in V$, el conjunto de aristas transformadas E' contiene las aristas *no dirigidas* v^1v^2 y v^2v^3. Además, para cada arista *dirigida* $vw \in E$, E' contiene la arista *no dirigida* v^3w^1. Dicho de otro modo, cada vértice de G se expande a tres vértices conectados por dos aristas, y una arista vw en E se convierte en una arista que va del tercer vértice correspondiente a v al primero correspondiente a w. En la figura 13.3 se presenta una

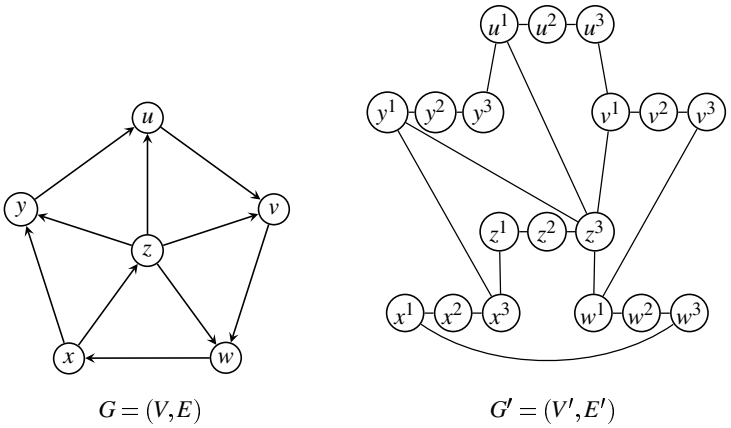


Figura 13.3 Reducción del problema del ciclo hamiltoniano dirigido al problema del ciclo hamiltoniano no dirigido.

ilustración. La transformación es directa, y es indudable que G' se puede construir en tiempo polinómicamente acotado. Si $|V| = n$ y $|E| = m$, entonces G' tendrá $3n$ vértices y $2n + m$ aristas.

Supóngase ahora que G tiene un ciclo hamiltoniano (dirigido) $v_1, v_2, \ldots, v_n$. (Es decir, $v_1, v_2, \ldots, v_n$ son distintos, y existen las aristas $v_i v_{i+1}$, para $1 \leq i < n$, así como la arista $v_n v_1$.) Entonces, $v_1^1, v_1^2, v_1^3, v_2^1, v_2^2, v_2^3, \ldots, v_n^1, v_n^2, v_n^3$ es un ciclo hamiltoniano no dirigido para G'. Por otra parte, si G' tiene un ciclo hamiltoniano no dirigido, los tres vértices v^1, v^2 y v^3 que corresponden a un vértice de G deberán recorrerse consecutivamente, en el orden v^1, v^2, v^3 o v^3, v^2, v^1, ya que no es posible llegar a v^2 desde ningún otro vértice de G'. Puesto que las otras aristas de G' conectan vértices con subíndices 1 y 3, si en cualquier tripleta el orden de los subíndices es 1, 2, 3, el orden será 1, 2, 3 en todas las tripletas; de lo contrario, será 3, 2, 1 en todas las tripletas. Dado que G' no está dirigido, podemos suponer que su ciclo hamiltoniano es $v_{i_1}^1, v_{i_1}^2, v_{i_1}^3, v_{i_2}^1, \ldots, v_{i_n}^1, v_{i_n}^2, v_{i_n}^3$. Entonces, $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ es un ciclo hamiltoniano dirigido para G. Por tanto, G tiene un ciclo hamiltoniano dirigido si y sólo si G' tiene un ciclo hamiltoniano no dirigido. □

Desde luego, es mucho más fácil ver que el grafo G' definido en la demostración es la transformación adecuada, que imaginar el G' correcto por principio de cuentas, así que haremos unas cuantas observaciones acerca de la forma en que se escogió G'. Para garantizar que un ciclo en G' corresponda a un ciclo en G, necesitamos simular la dirección de las aristas de G. Este objetivo sugiere dar a G' dos vértices, digamos v^1 y v^3, por cada v de G con la interpretación de que v^1 se usa para las aristas de G cuya cabeza es v y v^3 se usa para las aristas cuya cola es v. Entonces, siempre que v^1 y v^3 aparezcan consecutivamente en un ciclo en G' se les podrá sustituir por v para obtener un ciclo en G, y viceversa. Lo malo es que ninguna característica de G' obliga a v^1 y v^3 a aparecer consecutivamente en todos sus ciclos; por ello, G' podría tener un ciclo hamiltoniano sin correspondencia con uno en G (véase el ejercicio 13.13). Se introduce el tercer vértice, v^2, al cual sólo se puede llegar desde v^1 o desde v^3, para forzar a los vértices que corresponden a v a que aparezcan juntos en cualquier ciclo en G'.

Teorema 13.8 El problema de la sumatoria de subconjunto (problema 13.5) se puede reducir al problema de calendarización de trabajos (problema 13.2).

Demostración Sea $s_1, \ldots, s_n, C$ una entrada I para el problema de la sumatoria de subconjunto (que pregunta si existe un subconjunto de los objetos cuya suma sea exactamente C). Sea $S = \sum_{i-1}^{n} s_i$. Si $S < C$, la salida de I será *no*, e I podrá transformarse en cualquier entrada de calendarización de trabajos cuya salida sea *no*, por ejemplo, $t_i = 2$, $d_i = p_i = 1$ y $k = 0$. Si $S \geq C$, I se transformará en la entrada siguiente: $t_i = p_i = s_i$ y $d_i = C$ para $1 \leq i \leq n$, y $k = S - C$. Es evidente que la transformación en sí ocupa poco tiempo.

Supóngase ahora que la entrada de sumatoria de subconjunto produce la respuesta *sí*; es decir, existe un subconjunto J de $N = \{1, 2, \ldots, n\}$ tal que $\sum_{i \in J} s_i = C$. Sea entonces π cualquier permutación de N que hace que todos los trabajos cuyos índices estén en J se realicen antes que cualesquier trabajos cuyos índices estén en $N - J$. Los primeros $|J|$ trabajos se terminarán antes de que venza su plazo, pues $\sum_{i \in J} t_i = \sum_{i \in J} s_i = C$ y C es el plazo para todos los trabajos. El castigo para los trabajos restantes es

$$\sum_{i=|J|+1}^{n} p_{\pi(i)} = \sum_{i=|J|+1}^{n} s_{\pi(i)} = S - \sum_{i \in J} s_i = S - C = k.$$

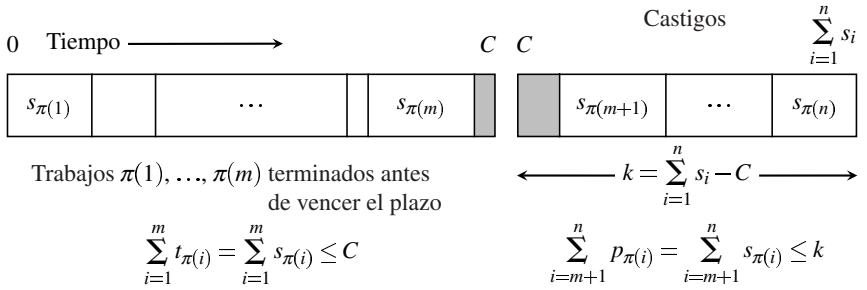


Figura 13.4 Un calendario de trabajos satisfactorio resuelve el problema de la sumatoria de subconjunto

Así pues, los trabajos se pueden realizar con un castigo total de k.

Por otro lado, sea π cualquier calendario para los trabajos cuyo castigo total es $\leq k$. Sea m el número de trabajos terminados antes de vencer el plazo común C; es decir, m es el número más grande tal que

$$\sum_{i=1}^{m} t_{\pi(i)} \leq C. \tag{13.1}$$

Entonces, el castigo es

$$\sum_{i=m+1}^{n} p_{\pi(i)} \leq k = S - C. \tag{13.2}$$

En la figura 13.4 se presenta una ilustración. Puesto que $t_i = p_i = s_i$ para $1 \leq i \leq n$, es necesario que

$$\sum_{i=1}^{m} t_{\pi(i)} + \sum_{i=m+1}^{n} p_{\pi(i)} = S,$$

y esto sólo puede suceder si las desigualdades de las ecuaciones 13.1 y 13.2 son igualdades (es decir, si las áreas sombreadas en la figura 13.4 son cero). Por tanto, $\sum_{i=1}^{m} t_{\pi(i)} = C$, y los objetos con índices $\pi(1), \ldots, \pi(m)$ son una solución del problema de la sumatoria de subconjunto. □

Presentaremos problemas de reducción similares en los ejercicios.

13.3.2 Algunos problemas $\mathcal{NP}$-completos conocidos

Reuniremos aquí varios problemas $\mathcal{NP}$-completos adicionales que se tratan en el capítulo y en los ejercicios.

Problema 13.10 Cobertura de vértices

Una *cobertura de vértices* para un grafo no dirigido G es un subconjunto C de vértices tal que toda arista incida sobre algún vértice en C. Veamos las aristas del grafo como un sistema irregular

de pasillos que se intersecan en vértices. ¿Cuántos guardias podemos apostar como mínimo en las intersecciones sin que algún pasillo no esté vigilado por al menos un guardia?

Problema de optimización: Dado un grafo no dirigido G, hallar una cobertura de vértices para G con el menor número posible de vértices.

Problema de decisión: Dado un grafo no dirigido G y un entero k, ¿tiene G una cobertura de vértices que consista en k vértices? ■

Problema 13.11 Camarilla

Una *camarilla* es un subconjunto K de vértices de un grafo no dirigido G tal que cada par de vértices distintos en K está unido por una arista de G. Dicho de otro modo, el subgrafo inducido por K es completo. Una camarilla de k vértices es una *k-camarilla*.

Cabe señalar que para colorear un grafo que tiene una k-camarilla se requieren al menos k colores.

Problema de optimización: Dado un grafo no dirigido G, hallar una camarilla con el mayor número posible de vértices.

Problema de decisión: Dado un grafo no dirigido G y un entero k, ¿tiene G una camarilla formada por k vértices? ■

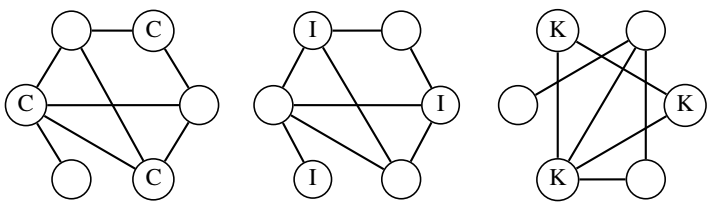
Problema 13.12 Conjunto independiente

Un *conjunto independiente* es un subconjunto I de vértices de un grafo no dirigido G tal que ningún par de vértices de I está unido por una arista de G.

Problema de optimización: Dado un grafo no dirigido G, hallar un conjunto independiente con el mayor número posible de vértices.

Problema de decisión: Dado un grafo no dirigido G y un entero k, ¿tiene G un conjunto independiente formado por k vértices? ■

Los tres problemas, cobertura de vértices, camarilla y conjunto independiente, están íntimamente relacionados, como sugiere la figura 13.5.



Cobertura de vértices Conjunto independiente Camarilla

Figura 13.5 Ejemplos de cobertura de vértices, conjunto independiente y camarilla

Problema 13.13 Conjunto de aristas de retroalimentación

Un *conjunto de aristas de retroalimentación* en un grafo dirigido G es un subconjunto F de aristas tal que todo ciclo en G tiene una arista en F.

Problema de optimización: Dado un grafo dirigido G, hallar un conjunto de aristas de retroalimentación con el menor número posible de aristas.

Problema de decisión: Dado un grafo dirigido G y un entero k, ¿tiene G un conjunto de aristas de retroalimentación formado por k aristas? ■

13.3.3 ¿Qué hace que un problema sea difícil?

Si restringimos de alguna manera el conjunto de entradas para un problema $\mathcal{NP}$-completo, el problema podría estar en $\mathcal{P}$; de hecho, podría tener una solución muy rápida. Técnicamente, restringir las entradas implica modificar la parte de pregunta del problema de modo que más ejemplares de entrada tengan respuestas *no* fáciles (es decir, en tiempo polinómico). Esto se hace añadiendo una condición a la pregunta estándar del problema, como ilustraremos con algunos ejemplos en la explicación que sigue. Es más útil ver la condición adicional como una restricción del conjunto de entradas.

No obstante, incluso con restricciones, el problema podría seguir siendo $\mathcal{NP}$-completo. Es importante conocer el efecto que restringir el conjunto de entradas de un problema que tiene sobre la complejidad porque, en muchas aplicaciones, las entradas que se dan en la realidad tienen propiedades especiales que podrían permitir una solución polinómicamente acotada. Lo malo es que los resultados no son alentadores; incluso con restricciones muy amplias de las entradas, muchos problemas $\mathcal{NP}$-completos siguen siendo $\mathcal{NP}$-completos.

Por otra parte, en muchas situaciones que se dan en ingeniería se dispone de cierta flexibilidad en cuanto a la forma de definir un problema y el criterio de optimización exacto. Si un criterio produce un problema $\mathcal{NP}$-completo, es muy posible que un criterio alterno, totalmente aceptable, produzca un problema en $\mathcal{P}$. Por ello, conocer bien las características de los problemas difíciles puede resultar muy útil en situaciones prácticas.

Definición 13.8 Grado de vértice

En un grafo dirigido o no dirigido, el *grado* de un vértice es el número de aristas que inciden en él. El grado máximo de cualquier vértice de G se denota con $\Delta(G)$. En el caso de grafos dirigidos, el *grado de entrada* y el *grado de salida* de un vértice son el número de aristas que entran y que salen, respectivamente. ■

En los problemas de grafos podemos considerar restricciones sobre $\Delta(G)$. Es fácil probar la mayor parte de las propiedades de los grafos en grafos con $\Delta \leq 2$. En tales grafos, el problema del ciclo hamiltoniano se puede resolver en tiempo polinómico. (Es decir, la pregunta modificada "¿Es $\Delta(G) \leq 2$ *y* G tiene un ciclo hamiltoniano?", es fácil.) El problema de la k-coloreabilidad también se puede resolver en tiempo polinómico si $\Delta \leq 3$. (Es decir, la pregunta modificada "¿Es $\Delta(G) \leq 3$ *y* G puede colorearse con k colores?" se puede resolver en tiempo polinómico.) Sin embargo, el problema del ciclo hamiltoniano es $\mathcal{NP}$-completo incluso con grafos en los que $\Delta = 3$. En grafos con $\Delta \geq 4$, la k-coloreabilidad es $\mathcal{NP}$-completa. Así pues, no es la presencia de vértices con grado alto lo que hace que estos problemas sean difíciles.

Por otra parte, el problema de la camarilla (problema 13.11) está en $\mathcal{P}$ en el caso de grafos con $\Delta \leq d$ para cualquier constante d. (Un algoritmo simple que verifica todos los subconjuntos de $d + 1$ vértices se ejecuta en tiempo $O(n^{d+1})$.) Esto implica que, en este caso, *sí* son los vértices de grado alto los que hacen que el problema sea difícil.

Un grafo *plano* se puede dibujar en un plano de modo que ningunas dos aristas se intersequen. Estos grafos se dan en muchas aplicaciones, por lo que vale la pena saber qué tan difíciles pueden ser diversos problemas si las entradas se restringen a grafos planos. (Determinar si un grafo arbitrario es plano o no es en sí un problema importante; por fortuna, se sabe que está en $\mathcal{P}$. Los mejores algoritmos para probar planicidad son complicados pero se ejecutan en tiempo lineal.) El problema del camino hamiltoniano dirigido es $\mathcal{NP}$-completo incluso si se le restringe a grafos dirigidos planos.

El problema de la cobertura de vértices (problema 13.10) sigue siendo $\mathcal{NP}$-completo cuando se le restringe a grafos planos. En cambio, la planicidad simplifica el problema de la camarilla (problema 13.11). Si el grafo es plano el problema está en $\mathcal{P}$ porque un grafo plano no puede tener una camarilla con más de cuatro vértices.

La 3-coloreabilidad sigue siendo $\mathcal{NP}$-completa si los grafos son planos y el grado máximo es 4. En cambio, la 4-coloreabilidad de grafos planos sí es polinómica, porque *todos* los grafos planos son 4-coloreables (éste es el famoso teorema de los cuatro colores). Es decir, la pregunta modificada "¿G es un grafo plano *y* G puede colorearse con 4 colores?", siempre tiene la misma respuesta que "¿G es un grafo plano?". No es necesario hallar un 4-coloreado para contestar esta pregunta.

Una de las primeras restricciones que se estudiaron fue la 3-satisfactibilidad (problema 13.6), que no permite a las fórmulas tener más de tres literales por cláusula. La 3-satisfactibilidad es $\mathcal{NP}$-completa. El problema de la 2-satisfactibilidad prohíbe a las fórmulas tener más de dos literales por cláusula, y se puede resolver en tiempo polinómico.

Otro fenómeno interesante, ilustrado por algunos de los ejemplos que siguen, es que dos problemas que al parecer tienen planteamientos muy parecidos podrían diferir mucho en cuanto a complejidad; uno podría estar en $\mathcal{P}$ mientras que el otro es $\mathcal{NP}$-completo.

Aunque el problema de la cobertura de vértices (problema 13.10) es $\mathcal{NP}$-completo, su dual, el problema de la cobertura de aristas (¿Existe un conjunto de k aristas tal que cada vértice incida en al menos una de ellas?), está en $\mathcal{P}$.

En el capítulo 7 vimos que existen algoritmos eficientes para hallar el camino simple más corto entre dos vértices dados de un grafo. El problema del camino simple más largo es $\mathcal{NP}$-completo. (El planteamiento como problema de decisión de estos dos problemas incluye un entero k como entrada y pregunta si existe un camino con menos de k aristas o con más de k aristas, respectivamente.)

Determinar si un grafo es 2-coloreable es fácil; determinar si es 3-coloreable es $\mathcal{NP}$-completo.

Como ya dijimos, la 2-satisfactibilidad se puede resolver en tiempo polinómico. No obstante, consideremos esta variación del problema: dada una fórmula CNF con cuando más dos literales por cláusula, y dado un entero k, ¿existe una asignación de verdad para las variables que satisfaga al menos k cláusulas? El problema es $\mathcal{NP}$-completo.

El problema del conjunto de aristas de retroalimentación (problema 13.13) es $\mathcal{NP}$-completo, pero el mismo problema con grafos no dirigidos está en $\mathcal{P}$.

El problema de la calendarización de trabajos con castigos (problema 13.2) es $\mathcal{NP}$-completo, pero si omitimos los castigos y simplemente preguntamos si existe un calendario tal que no más de k trabajos terminen después de vencido su plazo, el problema está en $\mathcal{P}$. (En otras palabras, si el castigo por no cumplir con un plazo es 1, podemos reducir al mínimo este castigo en un tiempo polinómicamente acotado.)

Estos ejemplos no permiten hacer generalizaciones bonitas acerca de *por qué* un problema es $\mathcal{NP}$-completo. Persisten muchas preguntas pendientes en este campo, siendo desde luego la principal ¿$\mathcal{P} = \mathcal{NP}$?

13.3.4 Problemas de optimización y problemas de decisión

En nuestras descripciones de ejemplos de problemas $\mathcal{NP}$-completos en la sección 13.2.2 incluimos dos aspectos de los problemas de optimización: podríamos pedir el *valor* óptimo de la solución (por ejemplo, el número cromático de un grafo o el número mínimo de cajones en los que cabe un conjunto de objetos) o podríamos pedir una solución real (un coloreado del grafo, un empacamiento de los objetos) que logre el valor óptimo. Así pues, tenemos tres tipos de problemas:

1. Problema de decisión: ¿existe una solución que supere alguna cota dada?
2. Valor óptimo: ¿qué valor tiene una solución que sea la mejor posible?
3. Solución óptima: hallar una solución que logre el valor óptimo.

Es fácil ver que la lista está en orden de dificultad creciente. Por ejemplo, si tenemos un coloreado óptimo de un grafo, basta con contar los colores para determinar el número cromático del grafo, y si conocemos su número cromático resulta trivial determinar si el grafo es *k*-coloreable para cualquier *k* dada. En aplicaciones reales generalmente nos interesa una solución óptima (o casi óptima).

Ha sido más fácil precisar la teoría de la calidad de $\mathcal{NP}$-completo en el caso de los problemas de decisión, y dado que los problemas de optimización son por lo menos tan difíciles de resolver como los problemas de decisión correspondientes, no hemos perdido nada esencial al hacerlo. Es decir, nuestros comentarios acerca de la dificultad de los problemas de decisión $\mathcal{NP}$-completos son válidos para los problemas de optimización asociados a ellos. Estos problemas de optimización suelen calificarse como $\mathcal{NP}$-difíciles, aunque no sean problemas de decisión. De hecho, en cierto sentido son más difíciles que los problemas de decisión $\mathcal{NP}$-completos porque no se sabe si están en $\mathcal{NP}$. Es decir, no se conoce ningún algoritmo de verificación polinómico capaz de determinar si una solución propuesta es *óptima* o no.

No obstante, supóngase que se descubre que $\mathcal{P} = \mathcal{NP}$. Si tuviéramos algoritmos en tiempo polinómico para resolver los problemas de decisión, ¿podríamos entonces hallar el valor de la solución óptima en tiempo polinómico? En muchos casos es fácil ver que sí podríamos. Consideremos el coloreado de grafos. Supóngase que tenemos un subprograma de función booleana `sePuedeColorear(G, k)` que se ejecuta en tiempo polinómico y devuelve **`true`** si y sólo si el grafo *G* se puede colorear con *G* colores. Entonces podríamos escribir el programa siguiente:

```
numeroCromatico(G)
    for (k = 1; k <= n; k++)
        if (sePuedeColorear(G, k))
            break;
    return k;
```

Puesto que cualquier grafo de *n* vértices se puede colorear con *n* colores, sabemos que `sePuedeColorear(G, k)` produce **`true`** con cuando más *n* iteraciones del ciclo **`for`**. Por tanto, si `sePuedeColorear` se ejecuta en tiempo polinómico, también lo hará el programa completo.

La misma técnica muestra que, también con otros problemas, si podemos resolver el problema de decisión en tiempo polinómico, podremos hallar el *valor* de la solución óptima en tiempo polinómico. No obstante, la cosa no siempre es tan sencilla. Consideremos el problema del vendedor viajero. Se nos da un grafo completo, habiéndose asignado un costo entero a cada una de sus aristas, y queremos determinar el costo de un recorrido mínimo, o ciclo hamiltoniano. Si `cotaPVV(G, k)` es una función que devuelve **`true`** si y sólo si existe un recorrido que no cuesta más de k, el programa siguiente determina el costo de un recorrido mínimo:

```
minPVV(G)
   for (k = 1; k <= ∞; k ++)
       if (cotaPVV(G, k))
          break;
   return k;
```

¿Cuántas iteraciones del ciclo pueden efectuarse? Sea W el máximo de los pesos de las aristas. Puesto que un ciclo hamiltoniano tiene n aristas, el peso de un recorrido mínimo es cuando más nW, así que no se efectuarán más de nW iteraciones. Lamentablemente, como indica la explicación del tamaño de las entradas en la sección 13.2.5, esto no basta para concluir que el programa se ejecuta en tiempo polinómico. Dejamos para los ejercicios demostrar que este programa se puede modificar para hallar el costo de un recorrido mínimo en tiempo polinómico y que, de hecho, se puede hallar un recorrido óptimo en tiempo polinómico; en ambos casos, claro, suponiendo que el problema de decisión tenga una solución en tiempo polinómico (véase el ejercicio 13.59).

13.4 Algoritmos de aproximación

Muchos centenares de problemas con aplicaciones importantes son $\mathcal{NP}$-completos. ¿Qué podemos hacer si necesitamos resolver uno de estos problemas? Hay varias estrategias posibles. Aunque no exista ningún algoritmo polinómicamente acotado, podría haber diferencias importantes entre las complejidades de los algoritmos conocidos; podemos tratar, como siempre, de desarrollar el más eficiente que se pueda. Podríamos concentrarnos en el comportamiento promedio, no en el de peor caso, para luego buscar algoritmos que sean mejores que otros según ese criterio. O bien, lo cual es más realista, podríamos buscar algoritmos que aparentemente funcionen bien con las entradas más comunes; esta decisión podría depender más de pruebas empíricas que de un análisis riguroso.

En esta sección estudiaremos un enfoque distinto para resolver problemas de optimización $\mathcal{NP}$-completos: el uso de algoritmos rápidos (es decir, polinómicamente acotados) que no garantizan la mejor solución pero sí una cercana a la óptima. Tales algoritmos se llaman *algoritmos de aproximación* o *algoritmos heurísticos*. Una heurística es una "regla práctica", casi siempre una idea que parece lógica, aunque no pueda demostrarse su bondad.

En muchas aplicaciones es suficiente con una solución aproximada, sobre todo si se toma en cuenta el tiempo necesario para hallar una solución óptima. Por ejemplo, de nada sirve hallar un calendario de trabajos óptimo si el costo del tiempo de computadora necesario para hallarlo excede el peor castigo que pudiera cobrarse.

Las estrategias o heurísticas, como suele llamárseles, que usan muchos algoritmos de aproximación son sencillas y directas, pero con algunos problemas podrían dar resultados sorprendentemente buenos. Muchas de ellas son heurísticas *codiciosas*. En el capítulo 8 estudiamos varios

algoritmos codiciosos que producen soluciones óptimas; en este capítulo no lo hacen. Para describir con precisión el comportamiento de un algoritmo de aproximación (qué tan buenos son sus resultados, no cuánto tiempo tarda) necesitamos varias definiciones. En los párrafos que siguen supondremos que estamos considerando un problema de optimización específico **P** y una entrada específica *I*.

Definición 13.9 Conjunto de soluciones factibles

Una *solución factible* es un objeto del tipo correcto pero no necesariamente óptimo. $FS(I)$ es el conjunto de soluciones factibles para I. ■

Ejemplo 13.3 Conjuntos de soluciones factibles

Para el problema de coloreado de grafos y un grafo de entrada G, $FS(G)$ es el conjunto de todos los coloreados válidos de G empleando cualquier cantidad de colores.

Para el problema de llenado de cajones con una entrada $I = \{s_1, \ldots, s_n\}$, $FS(I)$ es el conjunto de todos los empacados válidos empleando cualquier cantidad de cajones (es decir, todas las particiones de I en subconjuntos disjuntos $T_1, \ldots, T_p$, para alguna p, tal que el total de los s_i en cualquier subconjunto sea cuando más 1).

El conjunto de soluciones factibles para una entrada al problema de calendarización de trabajos es el conjunto de permutaciones de los n trabajos. ■

Definición 13.10 Función de valor

La función $val(I, x)$ devuelve el valor del parámetro de optimización que se logra con la solución factible x, para el ejemplar de entrada I. ■

Ejemplo 13.4 Funciones de valor

1. Para los coloreados de grafos, $val(G, C)$ es el número de colores empleados por el coloreado C.
2. Para el llenado de cajones, si $T_1, \ldots, T_p$ es una partición factible de los objetos cuando la entrada es I, $val(I, (T_1, \ldots, T_p)) = p$, el número de cajones empleados.
3. Para la calendarización de trabajos, $val(I, \pi) = P_\pi$, el castigo del calendario π. ■

El lector no deberá tener problemas para identificar los conjuntos de soluciones factibles y las funciones de valor de solución para otros problemas de optimización.

Definición 13.11 Valor óptimo

Dependiendo del problema, nos interesa hallar una solución que minimice o bien maximice *val*; supondremos que "mejor" es "mín" o "máx", respectivamente. Entonces, $opt(I) = \text{mejor}\{val(I, x) \mid x \in FS(I)\}$. Es decir, es el mejor valor que se puede lograr con cualquier solución factible. Una *solución óptima para I* es una x en $FS(I)$ tal que $val(I, x) = opt(I)$. ■

Definición 13.12 Algoritmo de aproximación

Un *algoritmo de aproximación* para un problema es un algoritmo en tiempo polinómico que, con la entrada I, produce como salida un elemento de $FS(I)$. ■

Hay varias formas de describir la calidad de un algoritmo de aproximación. Por lo regular la más útil es examinar el cociente del valor de la salida del algoritmo entre el valor de una solución óptima (aunque a veces podría interesarnos estudiar la diferencia absoluta entre los dos). Sea **A** un algoritmo de aproximación. Denotamos con $\mathbf{A}(I)$ la solución factible que **A** escoge cuando la entrada es I. Definimos:

$$r_{\mathbf{A}}(I) = \frac{val(I, \mathbf{A}(I))}{opt(I)} \qquad \text{para problemas de minimización,} \tag{13.3}$$

$$r_{\mathbf{A}}(I) = \frac{opt(I)}{val(I, \mathbf{A}(I))} \qquad \text{para problemas de maximización.} \tag{13.4}$$

En ambos casos, $r_{\mathbf{A}}(I) \geq 1$. Para resumir el comportamiento de **A**, nos gustaría considerar el cociente de peor caso. Una vez más, tenemos varias opciones: podríamos considerar el cociente de peor caso para todas las entradas de cierto tamaño, o para todas las entradas que tienen cierto valor de solución óptimo, o para todas las entradas. Los enfoques útiles dependen del problema. Definimos las funciones siguientes:

$$R_{\mathbf{A}}(m) = \text{máx}\ \{r_{\mathbf{A}}(I) \mid I \text{ tal que } opt(I) = m\}, \tag{13.5}$$

$$S_{\mathbf{A}}(n) = \text{máx}\ \{r_{\mathbf{A}}(I) \mid I \text{ de tamaño } n\}. \tag{13.6}$$

Cabe señalar que $R_{\mathbf{A}}(m)$ podría ser infinito para alguna m. En el caso de algunos problemas, el cociente máximo no está bien definido; esto puede suceder cuando el conjunto de entradas considerado es infinito. Para algunos problemas existen algoritmos de aproximación para los que R y S son arbitrariamente cercanos a 1; en otros casos R y S están acotados por constantes pequeñas, y en otros más no se conocen algoritmos que garanticen soluciones razonablemente cercanas. Para algunos problemas es posible demostrar que hallar una solución casi óptima es tan difícil como hallar una solución óptima. Presentaremos algunos algoritmos de aproximación en las secciones que siguen.

13.5 Llenado de cajones

El problema del llenado de cajones es una simplificación de una clase de problemas que se presentan con frecuencia en la práctica: cómo empacar o almacenar objetos de diversos tamaños y formas desperdiciando el mínimo de espacio. Se trata de uno de los problemas más antiguos para los que se hallaron algoritmos polinómicos que garantizan una solución cuya diferencia respecto a la solución óptima es un factor constante. Puesto que el factor constante es muy pequeño, estos algoritmos de aproximación y sus variantes resultan muy útiles en la práctica.

Sea $S = (s_1, \ldots, s_n)$, donde $0 < s_i \leq 1$, para $1 \leq i \leq n$. El problema consiste en empacar $s_1, \ldots, s_n$ en el menor número posible de cajones, donde cada cajón tiene capacidad unitaria. Se puede hallar una solución óptima considerando todas las formas de dividir un conjunto de n cosas en n o menos subconjuntos, pero el número de particiones posibles es mayor que $(n/2)^{n/2}$.

13.5.1 La estrategia de primer ajuste decreciente

El algoritmo de aproximación que presentamos aquí utiliza una estrategia heurística codiciosa muy sencilla, llamada *primer ajuste decreciente*; su complejidad en términos de tiempo en el peor caso está en $\Theta(n^2)$ y produce soluciones aceptables. La estrategia simple de *primer ajuste* coloca

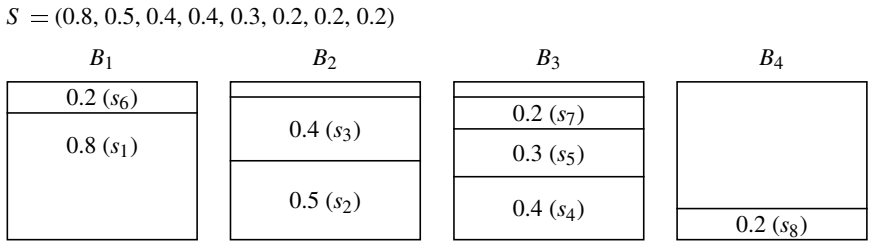


Figura 13.6 Ejemplo de heurística de *primer ajuste decreciente* para llenado de cajones: el empacado no es óptimo.

un objeto en el primer cajón en el que cabe. La estrategia de *primer ajuste decreciente* (*FFD*, *first fit decreasing* en inglés) es una modificación que primero ordena los objetos de modo que se consideren en orden no creciente por tamaño. Los tamaños no tienen que ser distintos, por lo que un nombre más correcto podría ser "primer ajuste no creciente", aunque "primer ajuste decreciente" es el nombre tradicional. En la figura 13.6 se presenta un ejemplo.

Algoritmo 13.1 Llenado de cajones: primer ajuste decreciente (FFD)

Entradas: Una sucesión $S = (s_1, \ldots, s_n)$ de tipo **`float`**, donde $0 < s_i \leq 1$ para $1 \leq i \leq n$. S representa los tamaños de los objetos $\{1, \ldots, n\}$ que se colocarán en cajones con capacidad 1.0 cada uno.

Salidas: Un arreglo `cajon` donde, para $1 \leq i \leq n$, `cajon[i]` es el número del cajón en el que se colocó el objeto i. Por sencillez, los objetos se indizan después de ordenarse en el algoritmo. El arreglo se pasa como parámetro y el algoritmo lo llena.

```
llenadoFFD(S, n, cajon)
    float[] usado = new float[n+1];
    // usado[j] es el espacio ya ocupado en el cajón j.
    int i, j;
    Inicializar todos los elementos de usado con 0.0.
    Ordenar S en orden descendiente (no creciente) para dar la sucesión
    s1 ≥ s2 ≥ ··· ≥ sn.

    for(i = 1; i <= n; i ++)
       // Buscar un cajón en el que quepa s[i].
       for (j = 1; j <= n; j ++)
          if (usado[j] + si <= 1.0)
             cajon[i] = j;
             usado[j] += si;
             break;   // salir de for (j)
       // Continuar for (i)
```

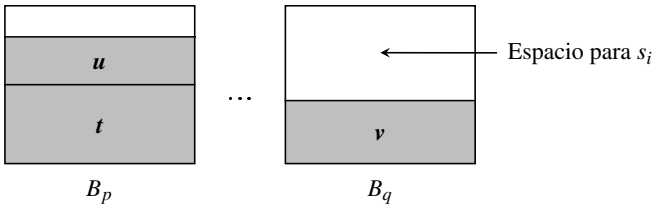


Figura 13.7 Primera ilustración de la demostración del lema 13.9

La entrada se puede ordenar en tiempo $\Theta(n \log n)$. El índice j se incrementa mientras se busca un cajón apropiado para s_i no más de $n(n-1)/2$ veces en total, contando todas las i. Todas las demás instrucciones se ejecutan cuando más n veces, así que la complejidad de peor caso en términos de tiempo está en $O(n^2)$.

La heurística FFD no siempre produce empacados óptimos; el empacado de la figura 13.6 no es óptimo. El teorema 13.11, que proporciona cotas superiores para los peores empacados producidos por FFD, se establece mediante los dos lemas siguientes. Después del teorema mencionaremos algunos resultados acerca del desempeño promedio de FFD.

Lema 13.9 Sea $S = (s_1, \ldots, s_n)$ una entrada, en orden no creciente, para el problema de llenado de cajones y sea $opt(S)$ el número óptimo (o sea, mínimo) de cajones para S. Todos los objetos colocados por FFD en cajones extra (es decir, cajones cuyo índice es mayor que $opt(S)$) tienen un tamaño no mayor que $1/3$.

Demostración Sea i el índice del primer objeto que FFD coloca en el cajón $opt(S) + 1$. Puesto que S está ordenado en orden no creciente, basta con demostrar que $s_i \leq 1/3$. Examinemos el contenido de los cajones en el momento en que FFD considera s_i. Supóngase que $s_i > 1/3$. Entonces, $s_1, \ldots, s_{i-1} > 1/3$, así que los cajones B_j para $1 \leq j \leq opt(S)$ contienen cuando más dos objetos cada uno. Afirmamos que, para alguna $k \geq 0$, los primeros k cajones contienen un objeto cada uno y los $opt(S) - k$ cajones restantes contienen dos cada uno. De lo contrario, existirían dos cajones B_p y B_q, como en la figura 13.7, con $p < q$, tales que B_p tiene dos objetos, digamos t y u (con $t \geq u$) y B_q sólo uno, v. Puesto que los objetos se consideran en orden no creciente, $t \geq v$ y $u \geq s_i$; por tanto, $1 \geq t + u \geq v + s_i$, y FFD habría colocado el objeto i en B_q.

Por tanto, FFD llena los cajones como se muestra en la figura 13.8. Puesto que FFD no colocó ninguno de los objetos $k + 1, \ldots, i$ en los primeros k cajones, ninguno de ellos puede caber. Por tanto, en una solución óptima habrá k cajones que no contienen ninguno de los objetos $k + 1, \ldots, i$; sin pérdida de generalidad, podemos suponer que ésos son los primeros k cajones. Entonces, en una solución óptima, aunque tal vez no estén dispuestos exactamente como en la figura 13.8, los objetos $k + 1, \ldots, i - 1$ estarán en los cajones $B_{k+1}, \ldots, B_{opt}$, y puesto que todos estos objetos tienen un tamaño de más de $1/3$, habrá dos en cada cajón y $s_i > 1/3$ no podrá caber. Sin embargo, una solución óptima debe colocar el objeto i en uno de los primeros $opt(S)$ cajones; por tanto, el supuesto de que $s_i > 1/3$ debe ser falso. □

Lema 13.10 Para cualquier entrada $S = (s_1, \ldots, s_n)$, el número de objetos que FFD coloca en gavetas extra es cuando más $opt(S) - 1$.

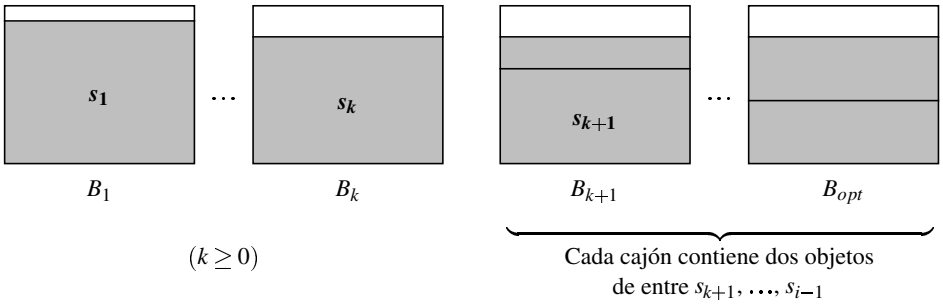


Figura 13.8 Segunda ilustración de la demostración del lema 13.9

Demostración Puesto que todos los objetos caben en $opt(S)$ cajones, $\sum_{i=1}^{n} s_i \leq opt(S)$. Supóngase que FFD coloca $opt(S)$ objetos con tamaños $t_1, \ldots, t_{opt(S)}$ en cajones extra y que b_j es el contenido final del cajón B_j, para $1 \leq j \leq opt(S)$. Si $b_j + t_j \leq 1$, FFD podría haber colocado t_j en B_j, así que

$$\sum_{i=1}^{n} s_i \geq \sum_{j=1}^{opt(S)} b_i + \sum_{j=1}^{opt(S)} t_i = \sum_{j=1}^{opt(S)} (b_i + t_i) > opt(S),$$

lo cual es imposible. □

Teorema 13.11 $R_{FFD}(m) \leq (4/3) + (1/3m)$. $S_{FFD}(n) \leq 3/2$ y, para infinitamente muchas n, $S_{FFD} = 3/2$.

Demostración Sea $S = (s_1, \ldots, s_n)$ una entrada con $opt(S) = m$. FFD no coloca en cajones extra más de $m - 1$ objetos, todos con tamaño menor que $1/3$, así que ocupa cuando más $m + \lceil (m - 1)/3 \rceil$ cajones. Por tanto,

$$r_{FFD}(S) \leq \frac{m + \lceil (m-1)/3 \rceil}{m} \leq 1 + \frac{m+1}{3m} \leq \frac{4}{3} + \frac{1}{3m}.$$

Así pues, $R_{FFD} \leq 4/3 + 1/3m$. Para una entrada de tamaño n, $r_{FFD}(S)$ es máximo con $m = 2$ (si $m = 1$, FFD sólo usa un cajón), así que $S_{FFD}(n) \leq 4/3 + 1/6 = 3/2$. La construcción de una sucesión de ejemplos I_n para una n arbitrariamente grande, donde $r_{FFD}(I_n) = 3/2$, se deja como ejercicio. □

Se conoce un resultado más categórico que el planteado en el teorema 13.11: el número de cajones extra que FFD usa está acotado por $2\,opt/9 + 4$, aproximadamente el 22% de la cantidad óptima. (Esto es, $R_{FFD}(m) \leq 11/9 + 4/m$.) Para una m arbitrariamente grande, hay ejemplos que demuestran que $R_{FFD}(m) \geq 11/9$, así que no podemos mejorar la cota para los peores empacados que FFD produce.

Por lo regular, FFD tiene un desempeño mucho mejor que el sugerido por estas cotas de peor caso. Se han efectuado extensos estudios empíricos con entradas grandes para determinar el número esperado (promedio) de cajones extra empleados por FFD (es decir, el exceso respecto al número óptimo requerido). Los datos se generaron aleatoriamente y con diversas distribuciones. El lector alerta quizá se pregunte cómo podrían efectuarse estudios extensos del número de cajones extra utilizados, con entradas grandes. ¿No necesitamos conocer el número óptimo de cajo-

nes para determinar el número de cajones extra empleados por FFD? ¡Estamos desarrollando algoritmos de aproximación precisamente porque se requiere demasiado tiempo para determinar el número óptimo de cajones cuando la entrada es grande! De hecho, los estudios empíricos no determinaron con exactitud el número óptimo de cajones; estimaron el número extra de cajones con base en la cantidad de espacio vacío que había en los empacados producidos por FFD. El espacio vacío es el número de gavetas utilizadas por FFD menos $\sum_{i=1}^{n} s_i$. Es obvio que el número de cajones extra empleados en un empacado está acotado por la cantidad de espacio vacío.

Con entradas S en las que $n = 128{,}000$ y tamaños de objetos distribuidos uniformemente entre 0 y 1, FFD produjo empacados que usan aproximadamente 64,000 cajones. La cota de peor caso más categórica (que ya mencionamos) garantiza que el número de cajones extra es cuando más $2\ opt(S)/9 + 4 \leq 2 \times 64{,}000/9 + 4 \approx 14{,}200$. De hecho, sólo quedaron cerca de 100 unidades de espacio vacío en los empacados FFD. Se ha demostrado que, con n objetos cuyos tamaños tienen una distribución uniforme entre 0 y 1, la cantidad esperada de espacio vacío en empacados FFD es de aproximadamente $0.3\sqrt{n}$. Por tanto, el número esperado de cajones extra no es mayor que aproximadamente $0.3\sqrt{n}$.

13.5.2 Otras heurísticas

Se puede usar la estrategia de primer ajuste (FF, *first fit* en inglés) sin ordenar los objetos. Los resultados no son tan buenos como con FFD, pero puede demostrarse que el número de cajones extra utilizados por FF no es más del 70% mayor que el óptimo (y algunos ejemplos alcanzan esa cifra). Estudios empíricos han demostrado que el comportamiento esperado de FF no es malo. Con $n = 128{,}000$, por ejemplo, el número de gavetas extra empleadas no fue más de aproximadamente el 2% del total de gavetas ocupadas.

Otra estrategia heurística codiciosa es el *mejor ajuste* (*BF*, *best fit* en inglés): un objeto de tamaño s se coloca en un cajón B_j, que es el más lleno de entre los cajones en que el objeto cabe; es decir, `usado[j]` es máximo sujeto al requisito de que `usado[j]` $+ s \leq 1.0$. Si los s_i se ordenan en orden no creciente, la estrategia de mejor ajuste tiene un desempeño similar al de FFD. Si los s_i no se ordenan, los resultados pueden ser peores pero el número de cajones seguiría siendo menor que dos veces el óptimo.

Existe otra estrategia aún más sencilla que FF y BF, la cual produce un algoritmo de aproximación más rápido y puede usarse en circunstancias en las que no es posible almacenar el contenido de todas las gavetas, debiéndose enviar a la salida conforme se efectúa el empacado. La estrategia se denomina *siguiente ajuste*. Los s_i no se ordenan, y se llena un cajón a la vez. Los objetos se colocan en el cajón actual hasta que el siguiente no cabe; entonces se inicia un cajón nuevo y ya no se colocan objetos en los cajones que ya se consideraron.

Ejemplo 13.5 Estrategia de siguiente ajuste

Sea $S = (0.2, 0.2, 0.7, 0.8, 0.3, 0.6, 0.3, 0.2, 0.6)$. Los objetos se colocarían en seis cajones, como en la figura 13.9, aunque cabrían en cuatro. ■

Es evidente que la estrategia de siguiente ajuste se puede implementar con un algoritmo de tiempo lineal. No obstante, parece probable que tal estrategia ocupe un gran número de cajones extra. Efectivamente, su comportamiento de peor caso es peor que el de FFD, pero la observación de que el total del contenido de dos cajones consecutivos cualesquiera debe ser mayor que 1 nos permite concluir que $R_{siguiente\ ajuste}(m) < 2$.

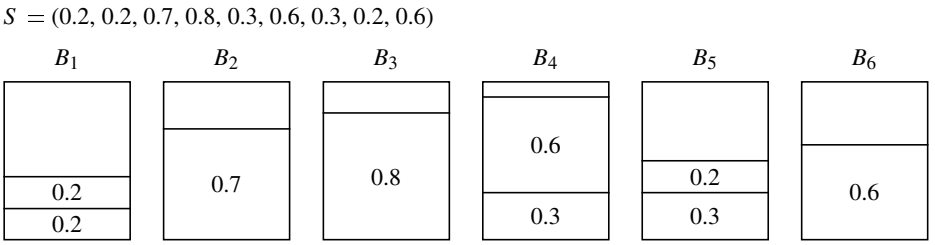


Figura 13.9 Ejemplo de la heurística de *siguiente ajuste* para llenado de cajones

Con algunas estrategias de llenado de cajones, si los s_i están acotados por algún número menor que 1, es posible demostrar mejores cotas (es decir, más bajas) para el cociente entre la salida real y la óptima.

13.6 Los problemas de la mochila y de la sumatoria de subconjunto

Una entrada del problema de la mochila (problema 13.4) consiste en un entero C y dos sucesiones de enteros, $(s_1, \ldots, s_n)$ y $(p_1, \ldots, p_n)$. Sea $N = \{1, \ldots, n\}$ el conjunto de índices. El problema consiste en hallar un subconjunto $T \subseteq N$ (T de "tomar") que maximice la utilidad total, $\sum_{i \in T} p_i$, sujeto a la restricción $\sum_{i \in T} s_i \leq C$; es decir, el tamaño total de los objetos tomados no es mayor que C.

Sea I una entrada específica del problema de la mochila. Utilizando la terminología y notación de la sección 13.4,

$$FS(I) = \left\{ T \mid T \subseteq N \quad \text{y} \quad \sum_{i \in T} s_i \leq C \right\}. \tag{13.7}$$

Dicho de otro modo, un algoritmo de aproximación deberá producir como salida un conjunto de objetos que quepa en la mochila, y cualquiera de esos T será una solución factible. La función de valor (definición 13.10) del problema de la mochila es

$$val(I, T) = \sum_{i \in T} p_i.$$

Es decir, la utilidad total de los objetos especificados por T. (De aquí en adelante omitiremos de *val* el parámetro I.) Podemos hallar una solución óptima calculando $val(T)$ para cada $T \subseteq N$, pero hay 2^n subconjuntos T.

Describiremos algunos algoritmos de aproximación para una versión un poco más sencilla del problema de la mochila, que equivale al planteamiento del problema de la sumatoria de subconjunto (problema 13.5) como problema de optimización. En el *problema de la mochila simplificado*, la utilidad de cada objeto es igual a su tamaño. Así pues, la entrada es un entero C y una sucesión $(s_1, s_2, \ldots, s_n)$. Queremos hallar un subconjunto $T \subseteq N$ que maximice $\sum_{i \in T} s_i$ sujeto al requisito de que $\sum_{i \in T} s_i \leq C$.

Los algoritmos que describiremos se pueden extender al problema general de la mochila partiendo de la lista de objetos ordenada por "densidad de utilidad"; es decir, ordenándola de modo que $p_1/s_1 \geq p_2/s_2 \geq \cdots \geq p_n/s_n$. En los algoritmos hay unos cuantos puntos en los que las referencias a tamaños tendrían que sustituirse por referencias a utilidades; deberá ser obvio cuáles son esos puntos. Los teoremas acerca de la cercanía de las aproximaciones también se pueden extender fácilmente al problema general de la mochila.

Existe una estrategia heurística codiciosa muy sencilla. Sea M el valor (utilidad) máximo de cualquier objeto de la entrada. Primero recorremos la sucesión de objetos y los vamos colocando en la mochila si caben. Sea V la sumatoria de los valores de los objetos escogidos. Ahora, si $V < M$, sacamos todo de la mochila y metemos un objeto de valor M. No es difícil demostrar que con esta estrategia la sumatoria de los objetos escogidos será por lo menos la mitad de la óptima; es decir, la razón definida en la ecuación (13.5) satisface $R_{codicioso}(m) \leq 2$ para toda $m > 0$. Podemos mejorar mucho esto.

Presentaremos una sucesión de algoritmos polinómicamente acotados $\texttt{mochilaS}_k$ para los cuales el cociente de la solución óptima entre la salida del algoritmo es $1 + 1/k$ (`mochilaS` significa "mochila simplificada"). Por tanto, podremos acercarnos a la solución óptima tanto como queramos. Sin embargo, la cantidad de trabajo efectuada por $\texttt{mochilaS}_k$ está en $O(kn^{k+1})$, así que cuanto más cercana sea la aproximación más alto será el grado del polinomio que describe la cota de tiempo. Utilizando la idea principal en estos algoritmos junto con un ardid adicional, es posible obtener una sucesión de algoritmos que produzcan resultados igual de buenos pero se ejecuten en tiempo $O(n + k^2n)$. (Véase Notas y referencias al final del capítulo.)

Para $k \geq 0$, el algoritmo $\texttt{mochilaS}_k$ considera cada subconjunto T que no tiene más de k elementos. Si $\sum_{i \in T} s_i \leq C$, recorrerá los objetos restantes (en algún orden arbitrario), $\{s_i \mid i \notin T\}$, y añadirá codiciosamente objetos a la mochila en tanto quepan. La salida es el conjunto así obtenido que produzca la sumatoria más grande. Presentaremos un ejemplo después del algoritmo.

Algoritmo 13.2 Aproximación de mochila simplificada $\texttt{mochilaS}_k$

Entradas: Un entero C y $s_1, s_2, \ldots, s_n$, una sucesión de enteros positivos.

Salidas: `tomar`, un subconjunto de $N = \{1, \ldots, n\}$; se pasa como parámetro un objeto que contendrá a `tomar`, y el algoritmo llena sus campos. Además, el algoritmo devuelve `sumaMax`, la sumatoria de s_i para $i \in$ `tomar`.

Comentario: Suponemos que se cuenta con la clase `ConjuntoIndices` para representar subconjuntos de N y que el conjunto de operaciones que se requieren tienen una implementación eficiente en esta clase. El parámetro de salida `tomar` pertenece a esta clase.

Procedimiento: Véase la figura 13.10. ■

Ejemplo 13.6 Aproximación de mochila simplificada

Supóngase que las entradas del problema son $C = 110$ y la sucesión (54, 45, 43, 29, 23, 21, 14, 1). Ya acomodamos la sucesión en orden descendiente para que sea más fácil trabajar con ella; esto no es un requisito del algoritmo. La tabla 13.1 muestra los subconjuntos considerados por $\texttt{mochilaS}_0$ y $\texttt{mochilaS}_1$. La solución óptima incluye los tamaños (43, 29, 23, 14, 1) y llena la mochila totalmente. $\texttt{mochilaS}_2$ hallaría esta solución. ■

```
mochilaS_k (C, S, tomar)
    int sumaMax, suma;
    ConjuntoIndices T = new ConjuntoIndices;
    int j;
    tomar = ∅; sumaMax = 0;
    Para cada subconjunto T ⊆ N con no más de k elementos:
        suma = ∑_{i∈T} s_i;
        if (suma <= C)
            // Considerar los objetos restantes.
            Para cada j que no esté en T:
                if (suma + s[j] <= C)
                        suma += s[j];
                        T = T ∪ {j};
            // Ver si T es el que mejor llena la mochila hasta ahora.
            if (sumaMax < suma)
                sumaMax = suma;
                Copiar los campos de T en tomar.
        // Continuar con el siguiente subconjunto de no más de k
            índices.
    return sumaMax;
```

Figura 13.10 Procedimiento para el algoritmo 13.2

	Subconjuntos de tamaño k	Objeto añadido por el ciclo **for** interior	suma
$k = 0$	0	54, 45, 1	100
	Objetos tomados: {54, 45, 1}	sumaMax =	100
$k = 1$	[54]	45,1	100
	[45}	54, 1	100
	[43}	54, 1	98
	{29}	54, 23, 1	107
	{23}	54, 29, 1	105
	{21}	54, 29, 1	105
	{14}	54, 29, 1	98
	{1}	54, 45	100
	Objetos tomados: {29, 54, 23, 1}	sumaMax =	107

Tabla 13.1 Ejemplo de mochila

Teorema 13.12 Para $k > 0$, el algoritmo $\texttt{mochilaS}_k$ efectúa $O(kn^{k+1})$ operaciones; $\texttt{mochilaS}_0$ efectúa $\Theta(n)$. Por tanto, $\texttt{mochilaS}_k \in P$ para $k \geq 0$.

Demostración Hay $\binom{n}{j}$ subconjuntos que contienen j elementos de N, así que el ciclo exterior se ejecuta $\sum_{j=0}^{k} \binom{n}{j}$ veces. Puesto que $\binom{n}{j} \leq n^j$ y $\binom{n}{0} = 1$, $\sum_{j=0}^{k} \binom{n}{j} \leq kn^k + 1$. La cantidad de trabajo que se efectúa durante una pasada por el ciclo está en $O(n)$, así que para todas las pasadas está en $O(kn^{k+1} + n)$. Corresponde al lector demostrar que el procesamiento extra para generar sistemáticamente un subconjunto con cuando más k elementos a partir del anterior se puede efectuar en tiempo $O(k)$ (ejercicio 13.37; el problema no es trivial). Por tanto, el trabajo total efectuado está en $O(kn^{k+1} + n)$, de lo que se sigue el teorema. □

Teorema 13.13 Para $k > 0$, $R_{\texttt{mochilaS}_k}(m)$ y $S_{\texttt{mochilaS}_k}(n)$, los cocientes de peor caso de la solución óptima entre el valor hallado por $\texttt{mochilaS}_k$ (ecuaciones 13.5 y 13.6) son cuando más $1 + 1/k$ para todos m y n.

Demostración Fijemos k y sean C y $s_1, \ldots, s_n$ una entrada específica I. Sea $opt(I) = m$. Supóngase que se obtiene una solución óptima llenando la mochila con p objetos con valores $s_{i_1}, s_{i_2}, \ldots, s_{i_p}$. Si $p \leq k$, este subconjunto será considerado explícitamente por $\texttt{mochilaS}_k$, así que $val(\texttt{mochilaS}_k(I)) = m$ y $r_{\texttt{mochilaS}_k}(I) = 1$. Consideremos ahora el caso en que $p > k$. El subconjunto formado por los k objetos más grandes de la solución óptima será considerado explícitamente como T por $\texttt{mochilaS}_k$. Sea j el primer índice de la solución óptima que $\texttt{mochilaS}_k$ no agrega a este T. (Si no existe tal j, quiere decir que $\texttt{mochilaS}_k$ da la solución óptima.) El objeto j no es uno de los k objetos más grandes incluidos en la solución óptima, cuya sumatoria es m, así que $m/(k + 1) \geq s_j$. Puesto que el objeto j fue rechazado, el espacio que todavía no se llena en la mochila es menor que s_j. Por tanto, $val(\texttt{mochilaS}_k(I)) + s_j > C \geq m$. La combinación de las últimas dos desigualdades da

$$val(\texttt{mochilaS}_k(I)) > m - \frac{m}{k+1} = \frac{mk}{k+1}.$$

Así pues, tenemos

$$r_{\texttt{mochilaS}_k}(I) = \frac{m}{val(\texttt{mochilaS}_k(I))} < (k+1)/k.$$

Puesto que esta cota es válida para cualquier entrada I, $R_{\texttt{mochilaS}_k}(m) \leq 1 + 1/k$ y $S_{\texttt{mochilaS}_k}(n) \leq 1 + 1/k$. □

Corolario 13.14 Dado cualquier $\epsilon > 0$, existe un algoritmo polinómicamente acotado $\mathbf{A}(\epsilon)$ que resuelve el problema de la mochila y para el cual $R_{\mathbf{A}(\epsilon)}(m) \leq 1 + \epsilon$ para toda $m > 0$, y $S_{\mathbf{A}(\epsilon)}(n) \leq 1 + \epsilon$ para toda n. □

Aunque existen algoritmos de aproximación para los que la razón $r(I)$ se puede hacer arbitrariamente cercana a 1, es muy poco probable que algún algoritmo de aproximación **A** pueda garantizar una cota de $O(1)$ para el error absoluto, que es $(opt(I) - val(I, \mathbf{A}(I)))$. Se puede demostrar que, si existe tal algoritmo, entonces $\mathcal{P} = \mathcal{NP}$. (La demostración no es muy difícil, y este problema se incluye en los ejercicios, pero recomendamos al lector leer la sección 13.7.2 antes de intentarla.)

13.7 Coloreado de grafos

Hemos hallado algoritmos de aproximación para los problemas de la mochila y de la sumatoria de subconjunto que producen resultados muy buenos; el cociente de los comportamientos para cualquier valor óptimo dado está acotado por una constante pequeña. Se han desarrollado varios algoritmos heurísticos para el problema de colorear un grafo, pero lamentablemente todos podrían producir coloreados muy alejados del óptimo. De hecho, se ha demostrado que si existiera un algoritmo de aproximación para colorear grafos que nunca usara más de aproximadamente dos veces el número óptimo de colores, sería posible obtener un coloreado óptimo en tiempo polinómicamente acotado, y ello implicaría $\mathcal{P} = \mathcal{NP}$. Por tanto, obtener coloreados cercanos al óptimo es tan difícil como obtener coloreados óptimos. (Demostraremos una versión un poco menos categórica de esta afirmación en la sección 13.7.2.)

13.7.1 Algunas técnicas básicas

En esta sección examinaremos una estrategia heurística codiciosa. Puede producir coloreados deficientes, pero resulta útil como subrutina en algoritmos más complejos que emplean menos colores. En la sección que sigue presentaremos un algoritmo así.

Sea $G = (V, E)$, donde $V = \{v_1, \ldots, v_n\}$, y sean los "colores" enteros positivos. La estrategia de *coloreado secuencial* (*SC*, por sus siglas en inglés) siempre colorea el siguiente vértice, digamos v_i, con el color más bajo que sea aceptable (es decir, el color más bajo que no se haya asignado ya a un vértice adyacente a v_i).

Algoritmo 13.3 Coloreado secuencial (SC)

Entradas: $G = (V, E)$, un grafo no dirigido, donde $V = \{v_1, \ldots, v_n\}$.

Salidas: Un coloreado de G.

```
colorSec(V, E)
    int c, i;
    for (i = 1; i <= n; i ++)
        for (c = 1; c <= n; c ++)
            Si ningún vértice adyacente a v_i tiene el color c:
                Colorear v_i con c.
                break;   // salir de for(c)
        // Continuar for (c)
    // Continuar for (i)
```

El algoritmo 13.3 se puede implementar fácilmente de modo que su complejidad de peor caso esté en $O(n^3)$.

El comportamiento de SC con un grafo dado depende del ordenamiento de los vértices. Para $k \geq 2$, definimos la sucesión de grafos $G_k = (V_k, E_k)$, donde $V_k = \{a_i, b_i \mid 1 \leq i \leq k\}$ y $E_k = \{a_i b_j \mid i \neq j\}$. En la figura 13.11 se presenta una ilustración. Si V se da en el orden $a_1, \ldots, a_k, b_1, \ldots, b_k$, entonces SC coloreará todos los a con un color y todos los b con otro, y el coloreado será óptimo. En cambio, si los vértices están ordenados $a_1, b_1, a_2, b_2, \ldots, a_k, b_k$, SC necesitará un color nuevo para cada para a_i y b_i, y usará un total de k colores. Así pues, $R_{SC}(2) = \infty$, y si tomamos $n = |V|$ como tamaño de un grafo, $S_{SC}(n) \geq n/4$ para $n \geq 4$.

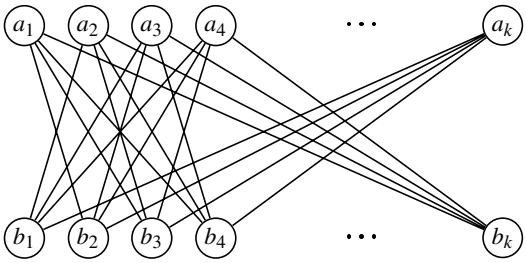


Figura 13.11 El grafo G_k

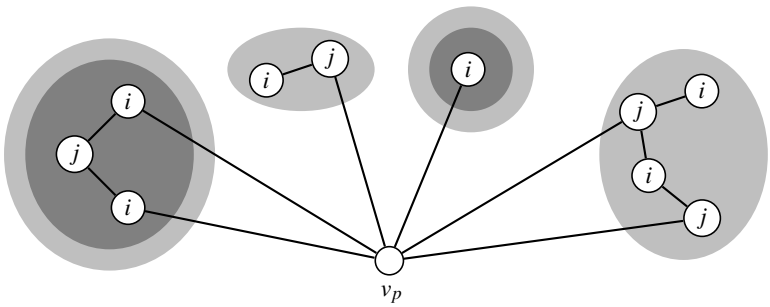


Figura 13.12 Intercambio de colores: G_{ij} consta de los cuatro componentes conectados que se encierran en gris claro. S_i consta de los dos componentes que también se encierran en gris oscuro.

Recordemos la definición 13.8, según la cual $\Delta(G)$ denota el grado máximo de cualquier vértice de un grafo G. Es fácil demostrar el teorema siguiente.

Teorema 13.15 El número de colores utilizados por el esquema de coloreado secuencial no es mayor que $\Delta(G) + 1$. □

Varios algoritmos para coloreado de grafos más complicados, que se basan en el coloreado secuencial, tienen características adicionales diseñadas para mejorar el comportamiento tan deficiente de SC. Una de esas características consiste en intercambiar dos colores en la porción coloreada del grafo si ello evita tener que usar un color nuevo. La regla de intercambio, que formularemos a continuación, se ilustra en la figura 13.12.

Supóngase que $v_1, \ldots, v_{p-1}$ se colorearon empleando los colores 1, 2, …, c (donde $c \geq 2$) y que v_p está adyacente a un vértice de cada color. Para cada par i y j, con $1 \leq i < j \leq c$, sea G_{ij} el subgrafo consistente en todos los vértices coloreados i o j y todas las aristas entre esos vértices. Si existe un par (i, j) tal que en cada componente conectado de G_{ij} todos los vértices adyacentes a v_p tengan el mismo color, se efectuará un intercambio. Cabe señalar que el subgrafo G_{ij} en sí es-

tá 2-coloreado, con los colores i y j. Si estos dos colores se intercambian en todo un componente conectado de G_{ij}, el resultado seguirá siendo un c-coloreado correcto de $v_1, \ldots, v_{p-1}$.

En términos específicos, sea S_i el conjunto de todos los vértices que están en componentes conectados de G_{ij}, donde los vértices adyacentes a v_p son de color i. Los colores i y j se intercambian en S_i. Ahora v_p está adyacente a vértices de color j en S_i, y v_p ya estaba adyacente a vértices color j en el resto de G_{ij}. Ahora v_p se colorea con i, y el algoritmo sigue con v_{p+1}. Este algoritmo se denomina *coloreado secuencial con intercambios* (*SCI*, por sus siglas en inglés).

El trabajo necesario para determinar cuándo intercambiar colores y efectuar el intercambio podría incrementar considerablemente el tiempo requerido por el algoritmo, pero los intercambios producirán mejores coloreados que SC con muchos grafos. Se obtendrán coloreados óptimos con los grafos G_k, con los cuales SC puede tener un desempeño deficiente (recomendamos al lector comprobar esto).

Recordemos a $\chi(G)$, el número cromático de G (definición 13.1). Se puede demostrar que SCI produce un coloreado óptimo con cualquier grafo G cuyo $\chi(G)$ sea 2. Sin embargo, para $k \geq 3$, existe una sucesión de grafos H_k, con $3k$ vértices, que son 3-coloreables, con los que SCI usa k colores: $H_k = (V_k, E_k)$, donde $V_k = \{a_i, b_i, c_i \mid 1 \leq i \leq k\}$ y $E_k = \{a_ib_j, a_ic_j, b_ic_j \mid i \neq j\}$. Así pues, $R_{SCI}(3) = \infty$, y $S_{SCI}(n) \geq n/9$ si n es lo bastante grande.

El lector podría percatarse de que, si los vértices de esta sucesión de grafos H_k están ordenados $a_1, \ldots, a_k, b_1, \ldots, b_k, c_1, \ldots, c_k$, entonces CSI produce un coloreado óptimo. Por tanto, otra forma de enfocar el problema de mejorar la estrategia de coloreado secuencial básica consiste en ordenar los vértices de una forma especial antes de asignar colores. Algunas de esas técnicas mejoran los coloreados que se obtienen con muchos grafos, aunque sigue habiendo casos en los que el desempeño es casi tan malo como el de SC y SCI.

13.7.2 El coloreado aproximado de grafos es difícil

No se conocen algoritmos de coloreo de grafos polinómicamente acotados para los que el cociente del número de colores empleados entre el número óptimo de colores esté acotado por una constante. De hecho, garantizar una cota constante pequeña para ese cociente es $\mathcal{NP}$-difícil.

Teorema 13.16 Si existiera una algoritmo de coloreado de grafos que se ejecuta en tiempo polinómico y que colorea todo grafo G con menos de $(4/3)\chi(G)$ colores, el problema de la 3-coloreabilidad se podría resolver en tiempo polinómico (y puesto que la 3-coloreabilidad es $\mathcal{NP}$-completa, podríamos concluir que $\mathcal{P} = \mathcal{NP}$).

Demostración Supóngase que G es una entrada para el problema de la 3-coloreabilidad; es decir, queremos saber si G se puede colorear con tres colores. Sea **A** un algoritmo de coloreado de grafos aproximado que se ajusta a la descripción dada en el teorema. Si **A** colorea G con tres colores, es obvio que G es 3-coloreable. Si **A** colorea G con cuatro o más colores, G no será 3-coloreable, pues 4 no es menor que $(4/3)3$. Por tanto, **A** usa tres colores en G si y sólo si G es 3-coloreable, y podríamos usar **A** para resolver el problema de la 3-coloreabilidad en tiempo polinómico. □

Lo único que hemos demostrado realmente es que no podemos aproximar un 3-coloreado con menos de cuatro colores, lo cual es una conclusión más bien limitada. No obstante, podemos demostrar un teorema similar incluso para grafos con número cromático grande. La demostración se vale de una construcción llamada *composición* de dos grafos. Informalmente, en la composición

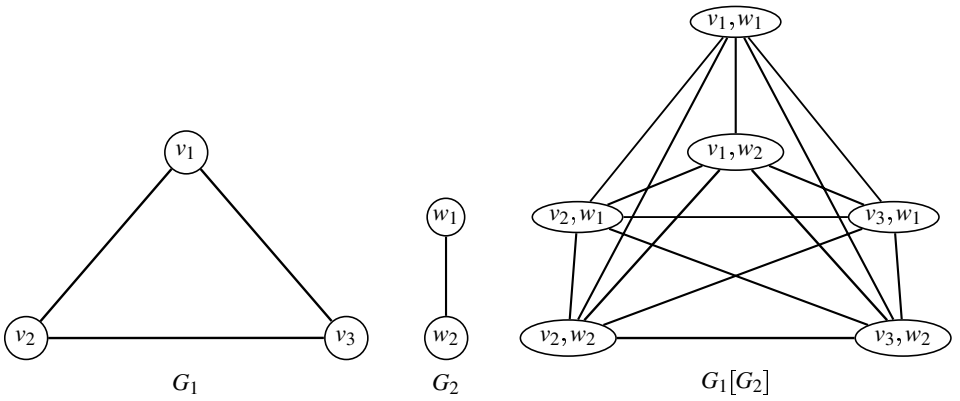


Figura 13.13 La composición de dos grafos

de G_1 y G_2, cada vértice de G_1 se sustituye por una copia de G_2. Una arista xy en G_1 se sustituye por aristas entre cada vértice de la copia de G_2 que sustituyó a x y cada vértice de la copia de G_2 que sustituyó a y. En la figura 13.13 se da un ejemplo. Ahora daremos la definición formal.

Definición 13.13 Composición de grafos

Sean $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ dos grafos. La *composición* de G_1 y G_2, denotada por $G_1[G_2]$, es el grafo $G = (V, E)$ en el que $V = V_1 \times V_2$ (es decir, pares ordenados con el primer componente tomado de V_1 y el segundo tomado de V_2). El conjunto de aristas es la unión de dos conjuntos, que llamamos aristas *locales* y aristas *de larga distancia*. Una arista local une dos vértices de la misma copia de G_2 y tiene la forma $(x, v)(x, w)$, donde $x \in V_1$ y $vw \in E_2$. Una arista de larga distancia une dos vértices de diferentes copias de G_2, donde las dos copias son "adyacentes" en términos de G_1; es decir, tiene la forma $(x, v)(y, w)$, donde $xy \in E_1$ y v y w son *cualesquier* vértices en V_2, no necesariamente distintos. ■

Es fácil ver que el número de vértices y aristas de G está acotado por polinomios en el número de vértices y aristas de G_1 y G_2, y que G se puede construir en tiempo polinómicamente acotado.

Teorema 13.17 Si existiera un algoritmo para colorear grafos en tiempo polinómico que usara menos de $(4/3)\chi(G)$ colores con cualquier grafo G para el cual $\chi(G) \geq k$, para algún entero k, el problema de la 3-coloreabilidad se podría resolver en tiempo polinómico.

Demostración Sea **A** un algoritmo que se ajusta a la descripción dada en el teorema. Sea G una entrada para el problema de la 3-coloreabilidad. Sea K_k el grafo completo con k vértices, y sea $H = K_k[G]$. H consta de k copias de G, donde cada vértice de una copia está conectado mediante una arista con cada uno de los vértices de cada una de las otras copias. Cada copia de G se puede colorear con $\chi(G)$ colores, pero dado que cada vértice de una copia está adyacente a todos los vértices de todas las demás copias, se necesita un conjunto nuevo de k colores para cada copia. Por tanto, $\chi(H) = k\chi(G)$. Puesto que esto es por lo menos k, se cumple la garantía de desempeño

de **A** en el caso de H. Ahora ejecutamos **A** con H y denotamos con $val(H, \mathbf{A}(H))$ el número de colores que **A** usa. Si G es 3-coloreable, entonces

$$val(H, \mathbf{A}(H)) < (4/3)\,\chi(H) \leq (4/3)3k = 4k.$$

Es decir, **A** usa menos de $4k$ colores. Por otra parte, si G no es 3-coloreable, necesita al menos cuatro colores y H necesita por lo menos $4k$ colores, así que **A** usa al menos $4k$ colores. Por tanto, podemos inferir si G es 3-coloreable o no ejecutando **A** con $H = K_k[G]$.

El tiempo de ejecución de **A** está acotado polinómicamente en términos del tamaño de H, y H se puede construir en tiempo polinómico a partir de G. Por tanto, ejecutar **A** con H y verificar si el número de colores empleados es o no menor que $4k$ contesta la pregunta acerca de la 3-coloreabilidad de G en tiempo polinómico. □

13.7.3 Algoritmo de Wigderson para colorear grafos

Como siempre, sea $G = (V, E)$ y $n = |V|$. Para muchas heurísticas de coloreado de grafos, el peor cociente del número de colores empleados entre el número óptimo puede estar en $\Theta(n)$; para otras, está en $\Theta(n/\log n)$. Durante mucho tiempo no se conocieron algoritmos mejores. Sin embargo, ahora hay uno (ideado por A. Wigderson) que tiene un desempeño un poco mejor (aunque todavía $R(3) = \infty$). El número de colores empleados está en $O(n^p)$ para $p < 1$ (pero p depende de $\chi(G)$). Si $\chi(G) = 3$, el algoritmo no usa más de $3\lceil\sqrt{n}\rceil$ colores.

Sea $v \in V$. La *vecindad* de v, denotada por $N(v)$, es el conjunto de vértices adyacentes a v. El subgrafo inducido por $N(v)$ se denota con $H(v)$; recordemos que consta de $N(v)$ y todas las aristas de G que unen vértices de $N(v)$. Cabe señalar que v no está en su vecindad.

Una idea clave del algoritmo es que los vecinos de vértices con grado alto se colorean primero. Mientras haya vértices de grado alto, los subgrafos de la vecindad se colorearán recursivamente (con los grafos 2-coloreables como fácil caso de frontera). Si todos los vértices tienen grado bajo, el grafo se colorea directamente. No es fácil seguir el algoritmo general y su análisis, por lo que primero presentaremos y analizaremos el algoritmo no recursivo de Wigderson para grafos 3-coloreables, y luego describiremos brevemente el algoritmo general.

Las vecindades y los subgrafos de vecindad, $N(v)$ y $H(v)$, dependen del grafo G. El algoritmo desecha vértices de G conforme los colorea; a medida que G cambia, también cambian las vecindades de los vértices. $N(v)$ y $H(v)$ siempre se definen en términos del grafo actual G.

El algoritmo utiliza el lema siguiente, que es fácil de demostrar.

Lema 13.18 Si G es k-coloreable entonces, para cualquier $v \in V$, $H(v)$ es $(k - 1)$-coloreable. □

Puesto que los grafos 2-coloreables se pueden identificar y colorear (con sólo dos colores) en tiempo polinómico, la vecindad de cualquier vértice de un grafo 3-coloreable se puede colorear con dos colores en tiempo polinómico.

Algoritmo 13.4 Coloreado aproximado de grafos 3-coloreables

Entradas: G, un grafo 3-coloreable; n, el número de vértices de G.

Salidas: Un coloreado de G.

```
color3(G)
    int c;   // el color actual
    c = 1;
    while (Δ(G) >= √n)
        Sea v un vértice de G con grado máximo.
        Colorear H(v) con los colores c y c + 1.
        Colorear v con el color c + 2.
        Borrar v y N(v) de G, y borrar todas las aristas que inciden en los vértices borrados.
        c += 2;
    // Ahora Δ(G) √n.
    Usar coloreado secuencial (SC) para colorear G, comenzando con el color c.
```

Ejemplo 13.7 `color3` en acción

Consideremos el grafo de la figura 13.14, en la que se explica la mayor parte de los pasos. Cabe señalar, empero, que el paso de coloreado secuencial podría haber usado tres colores, no dos. El coloreado que se produzca dependerá del orden en que se encuentren los vértices. Si, después de colorear con 3 el vértice que está en la parte más alta del grafo, el siguiente vértice encontrado fuera el que está hasta abajo, éste también se habría coloreado con 3 (porque estos dos vértices no están adyacentes); en tal caso se habría necesitado el color 5. ■

Teorema 13.19 Si G es 3-coloreable, `color3` produce un coloreado válido.

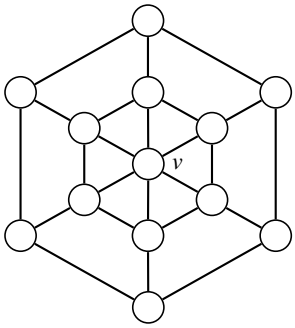
Demostración Por el lema 13.18, toda vecindad es 2-coloreable. Los colores empleados en $N(v)$ no se vuelven a usar, así que no pueden causar conflictos. El color empleado para colorear v sí se vuelve a usar, pero en el grafo que queda después de eliminarse $N(v)$, así que ningún otro vértice al que se le asigna el color de v está adyacente a v. Los colores empleados para colorear secuencialmente el grafo con $\Delta < \sqrt{n}$ no se vuelven a usar. □

Teorema 13.20 Si G es 3-coloreable, `color3` se ejecuta en tiempo polinómicamente acotado y usa cuando más $3\lceil\sqrt{n}\rceil$ colores.

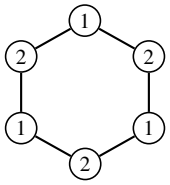
Demostración Primero, los tiempos. Puesto que todas las vecindades son 2-coloreables, todas se pueden colorear en tiempo polinómico (ejercicio 13.3). El algoritmo colorea vecindades en tanto $\Delta(G) \geq \sqrt{n}$. Así pues, para cada v cuya vecindad se colorea, $N(v)$ tiene por lo menos $\sqrt{n}$ vértices. Estos vértices se desechan después de colorearse, así que el número de iteraciones del ciclo **while** no puede ser mayor que $\sqrt{n}$. El coloreado secuencial (algoritmo 13.3) se ejecuta una vez después del ciclo **while**, y lo hace en tiempo polinómico. Por consiguiente, el trabajo total está polinómicamente acotado.

Se usan dos colores nuevos para cada vecindad coloreada en el ciclo **while** (cabe señalar que c se incrementa en 2 en ese ciclo). Por tanto, el ciclo usa cuando más $2\sqrt{n}$ colores para colorear todas las vecindades.

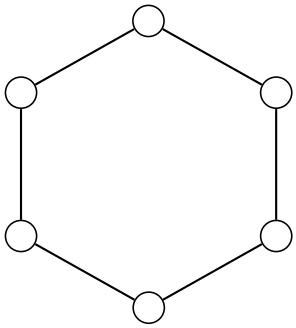
Cuando se usa el coloreado secuencial después del ciclo, $\Delta(G) < \sqrt{n}$, así que el número de colores empleados aquí no es mayor que $\Delta(G) + 1$ (por el teorema 13.15). Entonces, el coloreado secuencial usa cuando más $\lceil\sqrt{n}\rceil$ colores, y la cantidad total no es mayor que $3\lceil\sqrt{n}\rceil$. □



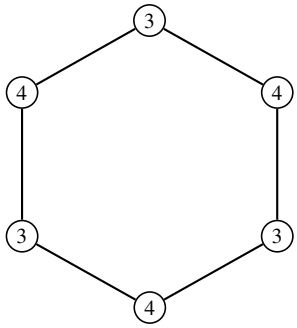
(a) Grafo G con $n = 13$ vértices: el grado de v es $6 \geq \sqrt{13}$



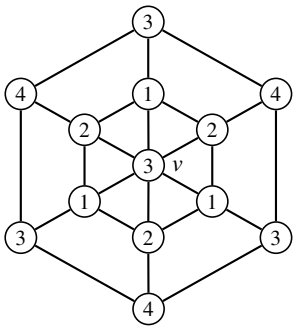
(b) $H(v)$: se asigna el color 3 a v



(c) G después de borrarse $H(v)$ y v: $\Delta(G) = 2 < \sqrt{13}$



(d) Coloreado secuencial



(e) El coloreado completo

Figura 13.14 Ejemplo para el algoritmo 13.4

En este punto, los lectores seguramente se están preguntando de qué sirve este algoritmo, ya que determinar si un grafo es 3-coloreable es un problema $\mathcal{NP}$-completo. De hecho, `color3` puede producir un coloreado válido con grafos de entrada para los que $\chi(G) > 3$; después de todo, puede usar hasta $3\lceil\sqrt{n}\rceil$ colores. El algoritmo `color3` se atasca si trata de 2-colorear un grafo de vecindad que no es 2-coloreable. Es fácil detectar tal fallo e informarlo. Por tanto, también es fácil modificar `color3` para que devuelva una variable booleana `coloreado` que indique si logró colorear su grafo de entrada o no. Los teoremas 13.19 y 13.20 se pueden generalizar para afirmar que `color3` siempre se ejecuta en tiempo polinómico y, si logra colorear el grafo de entrada (lo cual está garantizado si el grafo es 3-coloreable), producirá un coloreado válido empleando no más de $3\lceil\sqrt{n}\rceil$ colores.

★ El caso general

Ahora consideraremos el algoritmo de coloreado general. Recordemos que la idea clave era colorear primero vecindades de vértices de grado alto. Los subgrafos de vecindad se colorean recursivamente. ¿Qué tan pequeño debe ser $\Delta(G)$ para que efectuemos un coloreado directo en lugar de usar recursión? Este punto de corte se escoge de modo que equilibre aproximadamente el número de colores empleados por las partes recursiva y no recursiva. El valor empleado es $n^{1-1/(k-1)}$, donde k es un parámetro del algoritmo que podríamos ver como una conjetura del valor de $\chi(G)$. Sea $p(k) = 1 - 1/(k-1)$. En el caso de un grafo k-coloreable con n vértices, el algoritmo de coloreado recursivo, al que llamaremos `color`, se ejecuta en tiempo polinómico y produce un coloreado válido empleando cuando más $2k\lceil n^{p(k)}\rceil$ colores. (La demostración es un argumento más general, y más difícil, que utiliza las ideas empleadas en las demostraciones de los teoremas 13.19 y 13.20.)

Una vez más tenemos el problema de que no sabemos si un grafo arbitrario es o no k-coloreable. Queremos un algoritmo de coloreo que funcione bien con cualquier grafo, conozcamos o no su número cromático. Podemos usar `color` para obtener semejante algoritmo. Si k, la conjetura de $\chi(G)$, es demasiado pequeña, y `color` no puede colorear G, fallará en uno de los casos de "frontera"; es decir, la primera vez que intente 2-colorear un grafo que no es 2-coloreable. Por tanto, en este caso también podemos modificar `color` de modo que devuelva una variable booleana `coloreado` que indique si logró o no colorear su grafo de entrada. Ahora invocamos `color` repetidamente para hallar el valor más bajo de k con el que se logra colorear G. Para hallar tal k mínimo rápidamente, primero probamos sólo potencias de 2. Luego usamos un esquema tipo búsqueda binaria para verificar los valores entre dos potencias de 2 consecutivas. He aquí un bosquejo del esquema.

Algoritmo 13.5 Coloreado aproximado de grafos

```
colorAprox(G)
    k = 1;
    coloreado = falso;
    while (coloreado == falso)
       k = 2*k;
       coloreado = color(G, k);
    // El k0 mínimo con el cual color(G, k0) se ejecuta con éxito
    // está entre k/2 y k.
    Efectuar una búsqueda binaria en los enteros k/2, ..., k para hallar el k0 más pequeño
    tal que color(G, k0) devuelve true.
    Enviar a la salida el coloreado producido por color(G, k0).
```

Teorema 13.21 El algoritmo para colorear grafos aproximadamente, algoritmo 13.5, se ejecuta en tiempo polinómico y no emplea más de $2\chi(G)\lceil n^{1-1/(\chi(G)-1}\rceil$ colores.

Demostración El número de invocaciones de `color` en el algoritmo 13.5 (sin contar las invocaciones recursivas desde dentro de `color` mismo) es cuando más 2 lg k_0; dado que `color` se ejecuta en tiempo polinómico, el algoritmo 13.5 también se ejecuta en tiempo polinómico. Para toda $k \geq \chi(G)$, `color`(G, k) devuelve **`true`**, así que $k_0 \leq \chi(G)$. Por tanto, el número de colores empleados por el algoritmo 13.5 no es mayor que

$$2k_0\lceil n^{p(k_0)}\rceil \leq 2\chi(G)\lceil n^{p(\chi(G))}\rceil \leq 2\chi(G)\lceil n^{1-1/(\chi(G)-1)}\rceil. \quad \square$$

13.8 El problema del vendedor viajero

En el problema del vendedor viajero (TSP, por sus siglas en inglés) se nos da un grafo ponderado completo y se nos pide hallar un recorrido (un ciclo que pase por todos los vértices) con peso mínimo. Este problema tiene un gran número de aplicaciones en problemas de ruteo y calendarización. Por ello, tanto teóricos como prácticos han estudiado intensamente el problema. En esta sección presentaremos algunos algoritmos de aproximación fáciles, y luego plantearemos un teorema (sin demostrarlo) que dice que es poco probable que existan algoritmos de aproximación cuya bondad pueda demostrarse.

13.8.1 Estrategias codiciosas

En el capítulo 8 estudiamos dos algoritmos codiciosos para hallar árboles abarcantes mínimos en grafos ponderados no dirigidos (algoritmos de Prim y de Kruskal). Ambos algoritmos tienen variaciones naturales y fáciles para el problema del vendedor viajero. En esta sección investigaremos tales métodos.

Recordemos que el método codicioso para resolver problemas de optimización consiste en tomar decisiones sucesivas tales que cada decisión individual sea la mejor según cierto criterio limitado "a corto plazo" cuya evaluación es relativamente fácil. Una vez tomada una decisión, no se permite arrepentirse, ni siquiera si posteriormente queda claro que no fue una buena decisión. En general, las estrategias codiciosas son heurísticas: parecen lógicas, pero muchas de ellas no siempre dan pie a soluciones óptimas o no siempre son eficientes. En el capítulo 8 pudimos demostrar que las estrategias codiciosas de Prim y de Kruskal para resolver el problema del árbol abarcante mínimo *siempre* producen soluciones óptimas de manera eficiente.

Recordemos que el algoritmo de Prim principia en un vértice de inicio arbitrario y "cultiva" un árbol a partir de ese punto. En cada iteración del ciclo principal se escoge una arista que une un vértice de árbol con un vértice de borde; ello se hace "codiciosamente": se escoge la arista de menor peso.

El algoritmo de Kruskal, en cambio, toma "codiciosamente" la arista de menor peso que quede en cualquier parte del grafo, en tanto no forme un ciclo con las aristas que ya se escogieron. El subgrafo formado por las aristas ya escogidas en cualquier punto del algoritmo de Kruskal podría no ser conectado; es un bosque, pero no necesariamente un árbol (antes del final).

Las estrategias correspondientes para el problema del vendedor viajero se denominan estrategia del vecino más cercano y estrategia del eslabón más corto, respectivamente.

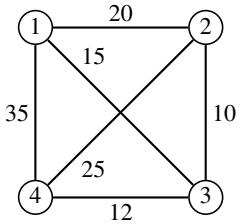


Figura 13.15 Una entrada para los algoritmos del vecino más cercano y del eslabón más corto

13.8.2 Estrategia del vecino más cercano

La *estrategia del vecino más cercano* es muy sencilla. En el algoritmo de Prim, cuando seleccionamos una arista nueva podríamos estar tomando una rama que sale de cualquier vértice del árbol. Aquí estamos construyendo un ciclo, no un árbol, así que siempre tomamos una rama que sale del extremo del camino que hemos construido hasta ese momento. Al final, añadimos una arista que une el último vértice con el vértice de inicio, para completar el recorrido. La estrategia del vecino más cercano podría describirse como sigue:

```
masCercanoTSP(V, E, W)
    Seleccionar un vértice arbitrario s para iniciar el ciclo C.
    v = s;
    Mientras haya vértices que no estén aún en C:
        Seleccionar una arista vw de peso mínimo, donde w no está en C.
        Añadir la arista vw a C;
        v = w;
    Añadir la arista vs a C.
    return C;
```

Es fácil implementar este algoritmo con un tiempo de peor caso en $O(n^2)$ para procesar un grafo de n vértices.

Si se ejecuta la estrategia del vecino más cercano con el grafo de la figura 13.15 partiendo del vértice 1, la salida es el ciclo 1, 3, 2, 4, 1, con un peso total de 85. ¿El algoritmo halló el recorrido mínimo? No. Éste es un ejemplo de estrategia codiciosa que no siempre da la solución óptima. Veamos si la estrategia del eslabón más corto es mejor.

13.8.3 Estrategia del eslabón más corto

Describiremos la *estrategia del eslabón más corto* para el caso de los grafos no dirigidos. Es preciso modificarla un poco si el grafo es dirigido (ejercicio 13.52). En cada iteración de su ciclo principal, la estrategia del eslabón más pequeño para el TSP (al igual que el algoritmo de Kruskal para el MST) toma una arista de peso mínimo de entre todas las aristas restantes en cualquier parte del grafo. Sin embargo, la estrategia del eslabón más corto desechará esa arista si no es posible que pueda formar parte de un recorrido junto con las otras aristas que ya se escogieron. El subgrafo formado por las aristas ya escogidas en cualquier punto del algoritmo constituye una co-

lección de caminos simples. No debe haber ciclos (antes del final) ni vértices en los que incidan más de dos aristas escogidas. El algoritmo termina una vez que ha procesado todas las aristas.

```
eslabonMasCortoTSP(V, E, W)
    R = E;  // R es las aristas restantes.
    C = ∅;  // C es las aristas del ciclo.
    Mientras R no esté vacío:
        Quitar la arista más ligera (más corta), vw, de R.
        Si vw no forma un ciclo con las aristas de C y vw no sería la tercera arista de C que
        incide en v o w:
            Añadir vw a C.
        // Continuar el ciclo
    Añadir la arista que conecta los extremos del camino que está en C.
    return C;
```

Podría hacerse que el ciclo **while** termine una vez que se hayan escogido $n - 1$ aristas. Es fácil mantener una cuenta de las aristas seleccionadas que inciden en cada vértice, así que el tiempo de ejecución de la estrategia del eslabón más corto es similar al del algoritmo de Kruskal.

Si se ejecuta la estrategia del eslabón más corto con el grafo de la figura 13.15, selecciona las aristas (2,3), (3,4), (1,2) y (1,4). El recorrido formado por esas aristas tiene un peso de 77, lo cual es mejor que el recorrido hallado por la estrategia del vecino más cercano, pero todavía no es óptimo. (Hállese el recorrido óptimo.)

13.8.4 ¿Qué tan buenos son los algoritmos de aproximación para el TSP?

No deberá sorprendernos que estas sencillas estrategias de tiempo polinómico para resolver el TSP no produzcan recorridos mínimos. Ya dijimos que el TSP es $\mathcal{NP}$-completo, y probablemente no exista ningún algoritmo que lo resuelva en tiempo polinómico. (Claro que esto no implica que las estrategias del vecino más cercano y del eslabón más corto siempre produzcan un recorrido no óptimo; a veces sí hallan el recorrido mínimo.)

Los algoritmos del vecino más cercano y del eslabón más corto son algoritmos de aproximación para el TSP. ¿Podemos establecer una cota para la diferencia entre los pesos de los recorridos hallados por estos algoritmos y el peso de un recorrido mínimo? Lamentablemente, no. Consideremos el teorema siguiente:

Teorema 13.22 Sea **A** cualquier algoritmo de aproximación para el problema del vendedor viajero (TSP). Si existe alguna constante c tal que $r_{\mathbf{A}}(I) \leq c$ para todos los ejemplares de entrada I, entonces $\mathcal{P} = \mathcal{NP}$.

Demostración Véase Notas y referencias al final del capítulo. □

Este teorema dice que es tan poco probable que existan algoritmos de aproximación con bondad "garantizada" para resolver el TSP como que existan algoritmos para resolver con exactitud el TSP en tiempo polinómico, aunque "bondad" se defina con holgura suficiente como para admitir cualquier múltiplo constante del peso de un recorrido mínimo. No obstante, si restringimos las entradas a grafos que tengan ciertas propiedades especiales, sí existen algoritmos de aproximación para el TSP con cotas para el peso de los recorridos producidos. Por ejemplo, si los pesos

de las aristas del grafo representan distancias en un plano, satisfacen la desigualdad del triángulo, que en este contexto es

$$W(u, w) \leq W(u, v) + W(v, w) \qquad \text{para todo } u, v, w \text{ en } G. \tag{13.8}$$

El ejercicio 13.53 describe un algoritmo de aproximación para el TSP que siempre produce un recorrido cuyo peso no es más del doble del óptimo si el grafo satisface la desigualdad del triángulo. Se conocen algoritmos con cotas aún mejores para esta clase especial de grafos.

13.9 Computación por ADN

Cuando oímos la palabra *computadora* pensamos en las computadoras electrónicas digitales modernas. Sin embargo, la palabra ha tenido muchos otros significados. Durante siglos, una computadora o computador fue una persona que se ganaba la vida efectuando cálculos o cómputos. La tecnología de computación ha evolucionado desde los dedos (para contar), pasando por diversos dispositivos mecánicos (el ábaco, las máquinas sumadoras, las ordenadoras de tarjetas), hasta las computadoras electrónicas de la era actual (desde macrocomputadoras o *mainframes* que ocupaban habitaciones enteras hasta computadoras personales sobre un escritorio, computadoras portátiles y sistemas incorporados). No hay motivo para creer que la evolución de la tecnología de cómputo se vaya a detener aquí. ¿Qué sigue? Una posibilidad son las computadoras basadas en ADN, o biológicas.

En 1994 Len Adleman, un computólogo que ya había adquirido fama por su papel en el desarrollo del sistema de cifrado con clave pública llamado RSA, demostró que un ejemplar de un problema $\mathcal{NP}$-completo podía resolverse empleando ADN. Ya vimos que es poco probable que existan algoritmos, en el sentido acostumbrado, para resolver en un tiempo razonable problemas $\mathcal{NP}$-completo. Los procesos bioquímicos operan con un gran número de moléculas en paralelo, lo que abre la posibilidad de obtener soluciones rápidas. El método de Adleman inauguró un enfoque totalmente nuevo hacia la computación, y ya se le está investigando intensamente. En esta sección describiremos el experimento de Adleman y luego analizaremos algunos trabajos recientes, así como el potencial y las limitaciones de los procesos bioquímicos utilizados por Adleman.

13.9.1 El problema y una perspectiva general del algoritmo

El problema que Adleman atacó es el del camino hamiltoniano en grafos dirigidos con vértices inicial y final designados. (Nos referiremos a este problema como HP (por *Hamiltonian Path*) en el resto de esta explicación.) Las entradas consisten en un grafo dirigido $G = (V, E)$, un vértice $v_{inicio} \in V$ y un vértice $v_{fin} \in V$. El problema de decisión consiste en determinar si existe un camino de v_{inicio} a v_{fin} que pase exactamente una vez por cada uno de los demás vértices de G. En muchas aplicaciones, si existe tal camino, nos gustaría hallarlo.

HP es $\mathcal{NP}$-completo. Por tanto, HP está en $\mathcal{NP}$ y, empleando la terminología de la sección 13.2.4, si se nos da una entrada para el problema y una solución propuesta, podremos verificar la validez de la solución en tiempo polinómico. Sea (G, v_{inicio}, v_{fin}) la entrada. Sea $n = |V|$. Sea $w_0, w_1, \ldots, w_q$ cualquier camino en G. Podemos verificar si se trata de un camino hamiltoniano de v_{inicio} a v_{fin} determinando si tiene las propiedades siguientes:

1. El camino principia y termina en los vértices correctos; es decir, $w_0 = v_{inicio}$ y $w_q = v_{fin}$.

2. El camino tiene la longitud correcta; es decir, $q = n - 1$.
3. Todos los vértices de V aparecen en el camino.

Estas verificaciones se pueden efectuar con gran rapidez (sin duda en tiempo polinómicamente acotado) con un algoritmo "normal". Como vimos en la sección 13.2.4, la capacidad para verificar un camino propuesto rápidamente no proporciona un algoritmo para resolver el problema del camino hamiltoniano en tiempo polinómico porque, en general, el número de caminos distintos a verificar no está acotado polinómicamente. No obstante, con la técnica del ADN, generamos hilos de ADN que representan caminos y los verificamos en paralelo. He aquí un resumen de alto nivel del algoritmo:

1. Generar hilos de ADN que representen caminos en G.
2. Usar procesos bioquímicos para extraer los hilos que satisfagan las propiedades 1 a 3 antes mencionadas, desechando todos los demás.
   a. Extraer hilos que principien en v_{inicio} y terminen en v_{fin}. (Desechar el resto.)
   b. Extraer hilos que incluyan n vértices. (Desechar el resto.)
   c. Extraer hilos que contengan a todos los vértices. (Desechar el resto.)
3. Cualquier hilo restante representa un camino hamiltoniano de v_{inicio} a v_{fin}. Si no queda ningún hilo, quiere decir que no existe tal camino en G.

Lamentablemente, los procesos bioquímicos no son tan exactos como las computadoras digitales. A medida que describamos el algoritmo dando más pormenores, mencionaremos los puntos en los que se presentan problemas. Más adelante analizaremos el impacto de esos problemas. Necesitamos saber un poco más acerca del ADN para entender cómo funciona el cómputo.

13.9.2 Generalidades del ADN

El ADN es el ácido desoxirribonucleico, el material genético que codifica las características de los seres vivos. Este breve panorama general sólo pretende dar las bases para entender cómo puede usarse el ADN para efectuar cálculos; desde el punto de vista de un biólogo, podría pecar de simpleza y falta de precisión.

El ADN consta de cadenas de compuestos químicos llamados nucleótidos. Hay cuatro nucleótidos en el ADN, que se denotan con la primera letra de su nombre: adenina (A), citosina (C), guanina (G) y timina (T). Podemos codificar cualquier información empleando este alfabeto de cuatro letras, así como podemos codificar cualquier información con bits (0 y 1). Actualmente es posible sintetizar hilos de ADN que contengan una sucesión específica de nucleótidos; es decir, crear cualquier cadena deseada de letras para representar datos.

John Watson y Francis Crick descubrieron la estructura de doble hélice del ADN (y fueron galardonados con el premio Nobel por sus trabajos). Los nucleótidos forman pares complementarios; A y T son complementos, así como C y G son complementos. Dos hilos de nucleótidos se unen entre sí (y se trenzan en una doble hélice) si tienen elementos complementarios en posiciones correspondientes. Véase por ejemplo la figura 13.16 (donde se ilustra la unión de hilos complementarios, pero no la doble hélice). El hecho de que hilos complementarios se unan se usa una y otra vez en el algoritmo por ADN para resolver el problema del camino hamiltoniano. Puede su-

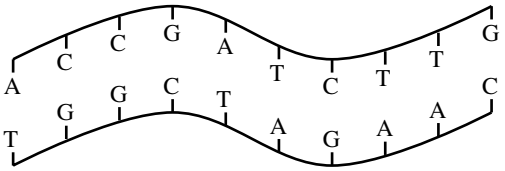


Figura 13.16 ADN de doble hilo en el que se muestran los pares complementarios

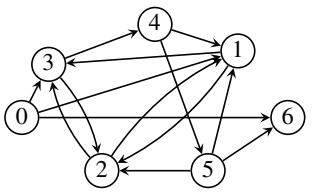


Figura 13.17 La entrada para el problema del camino hamiltoniano: $n = 7$, $v_{inicio} = v_0$, $v_{fin} = v_6$

ceder que dos hilos se unan aunque no tengan elementos complementarios en algunas posiciones; ésta es una de las propiedades de los procesos del ADN que pueden causar problemas con el algoritmo.

Kary Mullis, un químico, inventó un proceso llamado *reacción en cadena de polimerasa* (*PCR*, por sus siglas en inglés) que copia muestras pequeñas de ADN. (La PCR se usa ampliamente en investigaciones genéticas y forenses; Mullis también recibió el premio Nobel por sus trabajos.) La PCR se usa en varios pasos del algoritmo para reproducir hilos que satisfacen las propiedades de interés. Los procesos bioquímicos reales que se realizan en cada paso del algoritmo son complejos, pero no es preciso entenderlos para seguir la lógica del algoritmo. Por ello, nos basta con estos antecedentes.

13.9.3 El grafo dirigido de Adleman y el algoritmo por ADN

El grafo dirigido específico que Adleman usó como entrada para el problema se muestra en la figura 13.17.

Primero asociaremos una cadena R_i de 20 letras del alfabeto A, C, G, T a cada vértice v_i de G. Por ejemplo, R_2 = TATCGGATCGGTATATCCGA. Denotamos las letras de la cadena R_i con $d_{i,1}d_{i,2} \cdots d_{i,20}$. Entonces, $d_{2,1} = T$, $d_{2,6} = G$ y $d_{2,20} = A$.

Paso 1: Generar caminos en G

La "receta" para generar hilos de ADN que representen caminos en G utiliza dos tipos de ingredientes: hilos que representan aristas de G e hilos que representan vértices.

Primero describiremos los hilos que representan aristas de G. Para cada arista v_iv_j, tal que $v_i \neq v_{inicio}$ y $v_j \neq v_{fin}$, formamos un hilo, denotado por $S_{i \to j}$, empleando la segunda mitad de R_i y la primera mitad de R_j. Así pues, $S_{i \to j} = d_{i,11}d_{i,12} \cdots d_{i,20}d_{j,1}d_{j,2} \cdots d_{j,10}$.

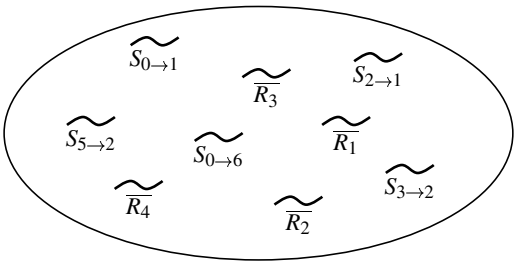


Figura 13.18 La "sopa" de ADN que contiene hilos de arista e hilos de vértice

Tomemos nota de que cada $S_{i\to j}$ tiene longitud 20, y que se conserva la orientación de las aristas en G. Es decir, $S_{i\to j} \neq S_{j\to i}$ (probablemente).

En el caso de aristas que salen del vértice de inicio o llegan al vértice final, creamos hilos un poco diferentes. Para formar esas aristas utilizamos la totalidad de R_{inicio} o R_{fin}. Por ejemplo, $S_{inicio\to3}$ consiste en la totalidad de R_{inicio} seguida de la primera mitad de R_3; es decir,

$$S_{inicio\to3} = d_{inicio,1}d_{inicio,2} \cdots d_{inicio,20}d_{3,1}d_{3,2} \cdots d_{3,10}.$$

Este hilo tiene longitud 30.

Se sintetiza un gran número de los hilos de arista, unas 10^{14} copias de cada uno para el grafo de 7 vértices y 14 aristas, y se colocan en la "cazuela".

Para cada vértice v_i, sin incluir v_{inicio} ni v_{fin}, se crea una gran cantidad (de nuevo unos 10^{14} para un grafo de este tamaño) de hilos que son el complemento de R_i; llamémoslos $\overline{R_i}$. Es decir, el nucleótido (letra) que está en cada posición de $\overline{R_i}$ es el complemento del nucleótido (letra) que está en la posición correspondiente de R_i. Por ejemplo, $\overline{R_2}$ es ATAGCCTAGCCATATAGGCT. Estos hilos se colocan en la cazuela junto con los de aristas. (La cazuela es en realidad un tubo de ensaye, y los ingredientes —el ADN junto con un poco de agua, sal y una sustancia llamada ligasa— ocupan un volumen de aproximadamente una cincuentava parte de una cucharadita, o un décimo de mililitro, para un grafo de este tamaño.) La figura 13.18 muestra algunos de los hilos de la mezcla.

Para crear hilos largos que representen caminos nos gustaría, por ejemplo, que se unieran $S_{4\to5}$, $S_{5\to2}$ y $S_{2\to1}$ (extremo a extremo) para representar el camino formado por las aristas v_4v_5, v_5v_2 y v_2v_1. ¿Qué podría hacer que se unieran estos hilos? Recordemos que los hilos de ADN se unen para formar dobles hilos si tienen elementos complementarios en posiciones correspondientes. Los hilos de vértice sostienen unidos los hilos de arista apropiados. Recordemos, por ejemplo, que la segunda mitad de $S_{5\to2}$ es la primera mitad de R_2 y que la primera mitad de $S_{2\to1}$ es la segunda mitad de R_2. Así pues, el hilo de vértice $\overline{R_2}$ se unirá a $S_{5\to2}$ y a $S_{2\to1}$, como se muestra en la figura 13.19. Ahora tenemos hilos (dobles) que representan caminos en G. Algunos de estos caminos son

$$v_4v_1v_2v_1 \qquad v_3v_2v_1 \qquad v_5v_6 \qquad v_0v_3v_4v_5v_6 \qquad v_0v_6$$
$$v_0v_1v_2v_3v_4v_5v_6 \qquad v_4v_5v_2v_1 \qquad v_0v_3v_2v_1v_2v_3v_4v_5v_6$$

La ligasa de la mezcla "pega" los hilos de arista unos a otros, de modo que las aristas que constituyen un camino sigan unidas después de que se eliminen los hilos de vértice en un paso subsecuente.

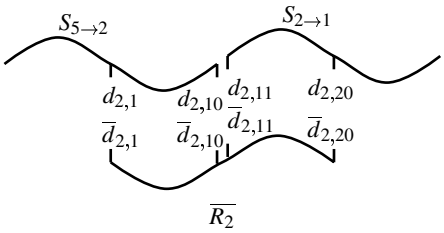


Figura 13.19 Unión de hilos para generar caminos

En este punto nos gustaría poder decir que ya tenemos hilos que representan *todos* los caminos simples de G, pero aquí radica uno de los problemas: es posible que no obtengamos todos los caminos simples. Aunque la probabilidad es muy pequeña (sólo hay unos cuantos centenares de caminos simples en este grafo), cabe la posibilidad de que los hilos necesarios simplemente no se hayan topado unos con otros para unirse. Por el momento haremos caso omiso de este problema y continuaremos. El próximo paso es eliminar los hilos que no satisfacen las propiedades 1 a 3 que describen los caminos hamiltonianos.

Paso 2a: Verificar que los vértices inicial y final son los correctos

Es posible hacer que el proceso PCR reproduzca hilos de ADN que tengan sucesiones específicas en los extremos de los hilos. En este caso se reprodujeron los hilos que comienzan con R_{inicio} y terminan con R_{fin}. Por tanto, consideramos que ahora la mezcla contiene hilos que representan caminos con los vértices inicial y final correctos. Algunos de esos caminos son

$$v_0v_6 \qquad v_0v_1v_2v_3v_4v_5v_6$$
$$v_0v_3v_2v_3v_4v_5v_6 \qquad v_0v_3v_2v_1v_2v_3v_4v_5v_6$$

Pese a que los hilos que representan estos caminos superan abrumadoramente en número a los hilos "malos" (es decir, los que representan caminos que no tienen los extremos correctos), es inevitable que queden algunos hilos "malos".

Paso 2b: Extraer los caminos de la longitud correcta

Una molécula de ADN que representa un camino tiene una copia completa de R_i por cada vértice v_i del camino. Cada R_i tiene longitud de 20. Nuestro grafo de entrada tiene siete vértices, así que nos interesa extraer los hilos de ADN de longitud 140. Existe un proceso para hacerlo. El ADN tiene carga negativa. La "sopa" de ADN se coloca en un extremo de un bloque de gel y se aplica una carga positiva al otro extremo del bloque. Las moléculas de ADN se desplazan hacia la carga positiva, pero las moléculas más pequeñas avanzan con mayor rapidez, lo que permite separar las moléculas de la longitud deseada. Una vez más, es posible que persistan algunos hilos no deseados. El proceso se repitió varias veces para reducir la fracción de hilos de longitud incorrecta.

Ahora tenemos hilos individuales de ADN que representan caminos como

$$v_0v_1v_2v_3v_4v_5v_6 \quad \text{y} \quad v_0v_3v_2v_3v_4v_5v_6.$$

Este ejemplo pone de manifiesto la necesidad del paso siguiente. El segundo camino tiene los vértices inicial y final correctos y la longitud correcta, pero pasa dos veces por v_3 y no pasa por v_1.

Paso 2c: Extraer los caminos que pasan por todos los vértices

Para cada vértice v_i (distinto de v_{inicio} y v_{fin}) por turno, agregamos copias de $\overline{R_i}$, extraemos los hilos a los que se unen, y desechamos los demás. $\overline{R_i}$ se unirá a los hilos que representen caminos que pasan por v_i. (Adleman unió las moléculas de $\overline{R_i}$ a gránulos magnéticos microscópicos, y luego usó un imán para separar de los otros los hilos deseados.) Luego se separan las moléculas de $\overline{R_i}$ de los hilos de camino y se eliminan. Ahora los hilos de camino restantes representan caminos que pasan por v_i. (Una vez más, es posible que se cuelen algunos hilos "malos".)

Una vez efectuado este paso con todos los vértices (distintos de v_{inicio} y v_{fin}), los hilos de ADN restantes, si es que queda alguno, representarán los caminos hamiltonianos deseados. La sucesión de nucleótidos del camino se puede leer con un dispositivo llamado secuenciador.

13.9.4 Análisis y evaluación

Corrección

El algoritmo teórico (generar todos los caminos, luego verificar las propiedades requeridas) es correcto pero, como ya señalamos, pueden darse "errores" en los procesos bioquímicos de la implementación por ADN. Es por ello que el cómputo por ADN no garantiza la respuesta correcta.

Todos los algoritmos que hemos estudiado en este libro funcionan correctamente con todas las entradas válidas (a menos que hayamos cometido un error de lógica). Por otra parte, existe una clase de algoritmos llamada *algoritmos probabilísticos* (programados para computadoras electrónicas ordinarias) que usan aleatoriedad en diversos pasos. Tales algoritmos podrían dar una respuesta incorrecta, o no dar una respuesta, o no darla dentro del plazo especificado. Los algoritmos probabilísticos programados para computadoras se pueden analizar matemáticamente. Podemos calcular la probabilidad de que produzcan la respuesta correcta. Tales algoritmos tienen ventajas que hacen que valga la pena sacrificar la "certidumbre" en algunas situaciones. Es común que sean mucho más rápidos que los algoritmos acostumbrados (deterministas) para resolver el mismo problema. En algunos casos, es posible determinar el equilibrio preciso entre el aumento en el tiempo de cómputo y el aumento en la probabilidad de que los resultados sean correctos. Algunos pueden diseñarse de modo que la probabilidad de un resultado erróneo sea más pequeña que la probabilidad de que ocurra un error de hardware en una computadora típica.

Los algoritmos por ADN son parecidos a los algoritmos probabilísticos. La ventaja potencial obvia en este caso es la rapidez que se logra por el hecho de que un número enorme de procesos bioquímicos se están llevando a cabo simultáneamente, en paralelo. En la actualidad los errores son una desventaja importante. Adleman halló el camino hamiltoniano gracias a un trabajo de laboratorio muy cuidadoso, repitiendo algunos de los procesos varias veces para purificar la solución de ADN. La utilidad práctica de este enfoque dependerá de trabajos futuros para mejorar las técnicas y reducir los errores de modo que la probabilidad de hallar las respuestas correctas sea elevada.

Análisis de tiempo y espacio

Resumiremos los pasos del experimento de Adleman, teniendo presente que contar pasos en el laboratorio es menos preciso que contar operaciones ejecutadas en una computadora digital. Sea $G = (V, E)$, $n = |V|$ y $m = |E|$.

1. Síntesis de hilos para los vértices y las aristas. El tiempo depende polinómicamente del tamaño del grafo.

2. Generación de caminos. Este paso depende del volumen de ADN, que depende del tamaño del problema. Los investigadores del campo piensan que este tiempo se puede considerar casi constante con volúmenes prácticos de material. Asimismo, el volumen de material procesado afecta el tiempo de los pasos subsiguientes, pero existen límites prácticos para el volumen de material, así que en cierto sentido los pasos tardan un tiempo constante. Sin embargo, falta conocer el tamaño máximo que puede tener un problema para poder resolverse con un volumen práctico de material. Volveremos después a esta pregunta.
3. Amplificación y extracción de hilos con los extremos deseados.
4. Extracción de hilos de la longitud deseada.
5. Para cada vértice (distinto de los extremos), extracción de los hilos que incluyen ese vértice. El número de pasos es proporcional al número de vértices.
6. En los pasos anteriores, varias aplicaciones de PCR, diversos procesos de lavado, calentamiento y otros.

Por tanto, hemos descrito una solución para un problema $\mathcal{NP}$-completo en un número lineal de pasos, pero los tiempos de los pasos dependen del volumen de material requerido para al entrada en cuestión. Con una cantidad fija de equipo de laboratorio, algunos de los pasos tardan un tiempo que es por lo menos lineal en términos de dicho volumen. Por tanto, para analizar la complejidad tanto en tiempo como en espacio es crucial entender cómo aumenta el volumen al aumentar el tamaño de las entradas.

Para el grafo de siete vértices, el volumen es aproximadamente 1/50 de cucharadita. ¿El volumen realmente será determinante en la práctica con entradas de tamaño razonable? Al principio del capítulo señalamos que los algoritmos con crecimiento exponencial dejan de ser prácticos con entradas de tamaño relativamente modesto. Si el volumen crece exponencialmente al aumentar el tamaño de las entradas, incluso un factor constante muy pequeño pronto será superado.

Restrinjámonos a grafos con grado de salida 2. El número de caminos con longitud $n - 1$ que parten del vértice de inicio es 2^{n-1}. (Recordemos que los caminos no tienen que ser simples.) Sin duda necesitaremos suficientes hilos como para generar al menos ese número de caminos, y en realidad necesitaremos generar muchos más. Con unos cuantos cálculos de tanteo podemos ver que si se requiere 1/50 de cucharadita (0.1 mL) para tener suficientes hilos para un grafo de siete vértices, se necesitarían unos 100,000 litros para un grafo de 37 vértices con grado de salida 2. Algunos investigadores han estimado que se necesitarían 10^{25} kg de nucleótidos para un grafo de 70 vértices. (Esto equivale aproximadamente a la masa de la Tierra.) Tal es la tiranía del crecimiento exponencial.

Este ejemplo pone de manifiesto la valía del análisis asintótico. Sin él, la gente podría gastar grandes cantidades de tiempo y dinero tratando de construir sistemas para resolver problemas mayores con métodos similares al empleado por Adleman. No obstante, Adleman y la comunidad de investigadores reconocen que se necesita algo más avanzado para que la computación por ADN pueda resolver problemas considerablemente más grandes. El propósito del experimento inicial fue determinar si se podía aprovechar el ADN para efectuar un cómputo importante, empleando la tecnología actual.

Direcciones futuras

La computación por ADN (y, en términos más generales, la computación molecular) es un campo muy activo. Una reseña exhaustiva rebasaría por mucho el alcance de esta obra. Recomenda-

mos a los lectores interesados consultar Notas y referencias al final del capítulo, y buscar literatura reciente. Algunas de las áreas de investigación son el control de errores, el mejoramiento de los factores constantes y el mejoramiento del orden asintótico.

La computación por ADN tiene algunas ventajas respecto a las computadoras electrónicas. Adleman resumió su potencial para lograr mayor rapidez, usar menos energía y almacenar datos de manera más densa. La velocidad de las computadoras mejora constantemente, por lo que las cifras que damos podrían haber perdido validez, aunque ilustran el punto. Cuando Adleman realizó su experimento (1994), las supercomputadoras más rápidas ejecutaban aproximadamente 10^{12} operaciones por segundo. Tomando la concatenación de moléculas de ADN (para generar caminos) como operación básica, Adleman estimó que el método de ADN efectuaba aproximadamente 10^{14} operaciones (en el curso de varias horas) y que esa cifra podía aumentarse a cerca de 10^{20}. A la velocidad más alta, el número de operaciones por segundo sería más de 1,000 veces mayor que la capacidad de una supercomputadora. Sin embargo, esta comparación debe interpretarse con cautela, porque todas las operaciones de la computadora están bajo la dirección de un programa, mientras que las de ADN están bajo un control tenue, y en gran medida son aleatorias.

El método del ADN consume menos energía que una supercomputadora. Adleman sugiere que el proceso de generación de caminos podría (en principio) efectuar más de 10^{19} operaciones por joule de energía, mientras que una supercomputadora ejecuta aproximadamente 10^{9} operaciones por joule. Un gramo de ADN, que ocupa cerca de un centímetro cúbico de espacio, puede almacenar tanta información como un billón de discos compactos.

En contraposición a la rapidez y el bajo consumo de energía de las operaciones moleculares está la dificultad para obtener las "salidas". El proceso real tardó siete días de tiempo real en un laboratorio: mucho tiempo para hallar un camino hamiltoniano en un grafo de siete vértices. Además, el experimento de Adleman requirió intervención y control humanos en todos los pasos. El proceso no estaba automatizado; no había "programa" que se introdujera en una máquina para ejecutarse. Falta que los investigadores hallen formas de automatizar el proceso.

Los algoritmos de ADN parecen naturalmente apropiados para problemas como el del camino hamiltoniano, porque es fácil ver la forma de representar caminos con hilos de ADN. ¿Las técnicas empleadas pueden aplicarse en general a muchos otros tipos de problemas? Se ha demostrado un resultado teórico fundamental acerca de la computación por ADN: utilizando unas cuantas operaciones básicas para recortar y pegar hilos de ADN, la computación por ADN es un *modelo universal de cómputo*. Esto significa que posee toda la potencia de cómputo de una computadora de aplicación general. Cualquier problema para el cual podamos escribir un algoritmo, en el sentido tradicional, que se ejecute en una computadora, se puede resolver empleando este modelo de computación por ADN, y es posible escribir programas en el ADN mismo.

Como ya vimos, la cantidad de material necesaria para generar todos los certificados de un problema con hilos de ADN puede crecer exponencialmente al crecer el tamaño de las entradas. Por tanto, el reto radica en hallar métodos cuyas necesidades de materiales (o sea, necesidades de espacio) no sean tan explosivas. Ciertos algoritmos por ADN más nuevos que se están desarrollando se valen de técnicas más avanzadas que generan algunas soluciones potenciales, eliminan las malas, generan más, y así, con lo que se reducen las necesidades de espacio.

La tecnología de computación por ADN es muy joven y los logros reales hasta ahora son pequeños. Se han efectuado cómputos en laboratorio con entradas tan pequeñas que, como el grafo de siete vértices de Adleman, se podrían haber resuelto con mucha mayor rapidez incluso sin usar una computadora. No obstante, es así como principia cualquier tecnología nueva. Las primeras computadoras electrónicas ocupaban grandes recintos y pesaban muchas toneladas, pero eran me-

nos potentes que computadoras que ahora podemos llevar en el bolsillo. Las investigaciones acerca de formas de acelerar los procesos químicos del ADN y hacerlos menos propensos a errores no se han suspendido. Parece probable que la computación por ADN resultará útil para resolver algunos tipos de problemas, sobre todo aquellos en cuya resolución se puede aprovechar el extenso paralelismo de los procesos bioquímicos. A estas alturas no sabemos qué tan útil será.

Ejercicios

Sección 13.2 *$\mathcal{P}$ y $\mathcal{NP}$*

13.1 Suponga que los algoritmos $\mathbf{A}_1$ y $\mathbf{A}_2$ tienen cotas de tiempo de peor caso p y q, respectivamente. Suponga también que el algoritmo $\mathbf{A}_3$ consiste en aplicar $\mathbf{A}_2$ a la salida de $\mathbf{A}_1$. (La entrada de $\mathbf{A}_3$ es la entrada de $\mathbf{A}_1$.) Dé una cota de tiempo de peor caso para $\mathbf{A}_3$.

13.2 Sugiera una condición necesaria y suficiente para que un grafo se pueda colorear con un color.

13.3 Escriba un algoritmo para determinar si un grafo $G = (V, E)$ es 2-coloreable o no. El algoritmo se deberá ejecutar en tiempo $\Theta(n + m)$, donde $n = |V|$ y $m = |E|$, y producir un 2-coloreado, si existe.

13.4 Demuestre que todos los problemas de decisión siguientes están en $\mathcal{NP}$. Para ello, describa una "solución propuesta" para un ejemplar del problema (en el sentido de la sección 13.2.4) e indique qué propiedades se verificarían para determinar si una solución propuesta justifica una respuesta de *sí* para el problema.

a. el problema del llenado de cajones

b. el problema del ciclo hamiltoniano

c. el problema de la satisfactibilidad

d. el problema de la cobertura de vértices (problema 13.10)

e. La pregunta: "¿El entero n es primo?" Tome nota de que no se puede suponer que las operaciones aritméticas tardan un tiempo $O(1)$ cuando los operandos son tan grandes como la entrada misma.

13.5

★ **a.** Sugiera una solución de programación dinámica para el problema de la sumatoria de subconjunto. (Vea también el ejercicio 10.21.) Analice el orden asintótico de su solución. Explique por qué esta solución no hace que el problema de la sumatoria de subconjunto esté en $\mathcal{P}$.

b. El ejemplo 10.3 presentó un procedimiento de programación dinámica para calcular el n-ésimo número de Fibonacci. Explique por qué ese procedimiento no se ejecuta en tiempo polinómico.

★★ **c.** Sugiera un algoritmo en tiempo polinómico para calcular el n-ésimo número de Fibonacci. Analice el orden asintótico de su algoritmo. *Sugerencia:* Considere el ejercicio 12.17.

Sección 13.3 Problemas $\mathcal{NP}$-completos

Nota: En el caso de los ejercicios en que se pide demostrar que un problema (**P**) se puede reducir a otro (**Q**), recuerde que ello implica varios pasos: definir una transformación de **P** a **Q** y demostrar que la transformación satisface las tres propiedades de la definición 13.6.

13.6 Demuestre que la reducción polinómica es una relación transitiva.

⋆ **13.7** Demuestre que la satisfactibilidad se puede reducir a la 3-satisfactibilidad. *Sugerencia:* La cláusula $C = (x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_k)$, donde $k \geq 4$, significa "(al menos) uno de $x_1, \ldots, x_k$ es verdad". Introduzca una variable nueva γ y escriba cláusulas que signifiquen "γ implica $x_1 \vee x_2$" y "$\neg\gamma$ implica $x_3 \vee \cdots \vee x_k$". ¿Cuántas literales tiene cada una de sus cláusulas? ¿Qué relación hay entre que C sea verdad y que cualquiera de estas dos cláusulas, o ambas, sean verdad?

13.8 El problema de la sumatoria de subconjunto se podría plantear de modo que $s_1, \ldots, s_n$ y C sean números racionales. Demuestre que esta versión del problema se puede reducir a la versión que se da en el texto, y viceversa.

13.9 El problema de la *intersección de conjuntos* se define como sigue:

Problema 13.14

Dados conjuntos finitos $A_1, A_2, \ldots, A_m$ y $B_1, B_2, \ldots, B_n$, ¿existe un conjunto T tal que

$$|T \cap A_i| \geq 1 \qquad \text{para } i = 1, 2, \ldots, m, \text{ y}$$
$$|T \cap B_j| \leq 1 \qquad \text{para } j = 1, 2, \ldots, n \text{ ?} \quad \blacksquare$$

Demuestre que el problema de la intersección de conjuntos es $\mathcal{NP}$-completo demostrando que está en $\mathcal{NP}$ y que la satisfactibilidad se puede reducir a él.

13.10 Demuestre que el problema del ciclo hamiltoniano para grafos no dirigidos se puede reducir al problema del ciclo hamiltoniano para grafos dirigidos.

13.11 Demuestre que el problema del ciclo hamiltoniano se puede reducir al problema del vendedor viajero. (Escoja grafos dirigidos o bien no dirigidos para ambos problemas.)

13.12 Demuestre que el problema del vendedor viajero es $\mathcal{NP}$-completo aunque los pesos se restrinjan a los valores $\{1, 2\}$. *Sugerencia:* Ello puede hacerse con una reducción del problema del ciclo hamiltoniano para grafos no dirigidos.

13.13 Suponga que un grafo dirigido $G = (V, E)$ se transforma en uno no dirigido $G' = (V', E')$, donde $V' = \{v^i \mid i = 1, 2 \text{ y } v \in V\}$ y $E' = \{v^1v^2 \mid v \in V\} \cup \{v^2w^1 \mid vw \in E\}$. Demuestre con un ejemplo que existe un grafo dirigido G tal que G no tiene un ciclo hamiltoniano pero G' sí.

13.14 Este problema considera un intento de transformación polinómica de un problema a otro que *no funciona*. Usted debe hallar el defecto. Un *grafo bipartita* es un grafo no dirigido en el que

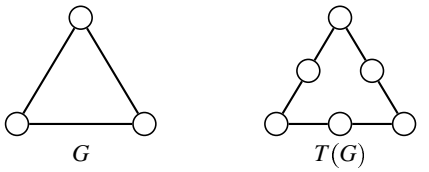


Figura 13.20 Transformación de un grafo en un grafo bipartita

todos los ciclos tienen longitud par. Intentamos demostrar que el problema del ciclo hamiltoniano (para grafos no dirigidos) se puede reducir al problema del ciclo hamiltoniano en grafos bipartitas. Necesitamos una función T : {grafos} $\rightarrow$ {grafos bipartitas} tal que T se pueda calcular en tiempo polinómico y, para todo grafo G, G tenga un ciclo hamiltoniano si y sólo si $T(G)$ tiene un ciclo hamiltoniano. Sea $T(G)$ el grafo bipartita que se obtiene insertando un vértice nuevo en todas las aristas. En la figura 13.20 se da un ejemplo. ¿Qué tiene de malo esta transformación?

13.15 Describimos una variación del problema del camino hamiltoniano dirigido en la que el camino debe iniciar y terminar en vértices específicos, digamos v_{inicio} y v_{fin}. En este ejercicio se demuestra que esta variación también es $\mathcal{NP}$-completa.

a. Demuestre que este problema está en $\mathcal{NP}$ describiendo brevemente un algoritmo que verifique un certificado para un ejemplar dado del problema.

★ **b.** Demuestre que el problema del ciclo hamiltoniano dirigido se puede reducir al problema del camino hamiltoniano dirigido con vértices inicial y final específicos.

★ **13.16** Demuestre que el problema de la 3-coloreabilidad se puede reducir al problema de la satisfactibilidad. (Esto, desde luego, es consecuencia del teorema de Cook; dé una transformación directa.)

13.17 Demuestre que el problema de la 3-coloreabilidad se puede reducir al problema de la 4-coloreabilidad.

13.18 Demuestre que el problema de decisión de la camarilla (problema 13.11) es $\mathcal{NP}$-completo mostrando que está en $\mathcal{NP}$ y utilizando después la transformación polinómica siguiente para reducir al satisfactibilidad a él. Suponga que $C_1, C_2, \ldots, C_p$ son las cláusulas de una fórmula CNF; las literales de la i-ésima cláusula se denotan con $l_{i,1}, l_{i,2}, \ldots, l_{i,q_i}$. La fórmula se transforma en el grafo con $V = \{(i, r) \mid 1 \leq i \leq p, 1 \leq r \leq q_i\}$; es decir, V tiene un vértice que representa cada ocurrencia de una literal en una cláusula y $E = \{(i, r)(j, s) \mid i \neq j$ y $l_{ir} \neq \overline{l_{j\,s}}\}$. Dicho de otro modo, existirá una arista entre dos vértices que representan literales de cláusulas distintas en tanto sea posible asignar a ambas literales el valor **`true`**. Sea $k = p$.

13.19 Si un grafo tiene una k-camarilla, es evidente que cualquier coloreado deberá usar al menos k-colores. Sin embargo, los k colores podrían ser insuficientes. Dé un ejemplo de grafo en el que el tamaño de camarilla más grande sea 3, pero se necesiten cuatro colores para colorear el grafo.

13.20 Demuestre que el problema de decisión de la camarilla (problema 13.11) se puede reducir al de la cobertura de vértices (problema 13.10).

13.21 Un *conjunto de vértices de retroalimentación* en un grafo dirigido $G = (V, E)$ es un subconjunto V' de V tal que V' contiene por lo menos un vértice de cada ciclo dirigido de G. El *problema del conjunto de vértices de retroalimentación* es:

Problema 13.15

Dado un grafo dirigido G y un entero k, ¿G tiene un conjunto de vértices de retroalimentación con por lo menos k vértices? ■

Demuestre que el problema de decisión de la cobertura de vértices (problema 13.10) se puede reducir al problema del conjunto de vértices de retroalimentación.

13.22 Considere el problema siguiente: una organización tiene 200 miembros y 17 comités. Cada comité se debe reunir toda una tarde durante la semana en la que se celebra la reunión anual de la organización. Se cuenta con una lista de los miembros de cada comité. Lo que hay que hacer es determinar si es posible programar las reuniones de los comités en cinco tardes de modo que cada miembro pueda asistir a la reunión de cada uno de los comités a los que pertenezca. ¿Cuál de los problemas que vimos en este capítulo es el que más se parece a éste? Explique la correspondencia.

13.23 Idee un algoritmo para determinar el número cromático de grafos que poseen la propiedad de que ningún vértice tiene grado mayor que 2 (es decir, en ningún vértice inciden más de dos aristas). El tiempo de ejecución de su algoritmo deberá ser lineal en términos del número de vértices del grafo.

12.24 Hemos dicho que el problema de la cobertura de vértices (problema 13.10) es $\mathcal{NP}$-completo. Demuestre que si las entradas se restringen a árboles (grafos no dirigidos, acíclicos, conectados), es posible hallar una cobertura de vértices mínima en tiempo polinómico. (Con una implementación cuidadosa, podrá idear un algoritmo lineal.)

⋆ **13.25** Idee un algoritmo polinómicamente acotado para determinar si una fórmula CNF con no más de dos literales por cláusula es satisfactible. Determine la complejidad de peor caso de su algoritmo. *Sugerencia:* Construya un grafo dirigido asociado a la fórmula; luego use un algoritmo del capítulo 7.

13.26 Escriba un algoritmo polinómicamente acotado para determinar si un grafo tiene una 4-camarilla. Determine la complejidad de peor caso de su algoritmo.

13.27 Dé condiciones necesarias y suficientes para que un grafo no dirigido con grado máximo de 2 tenga un ciclo hamiltoniano. Bosqueje un algoritmo eficiente para probar las condiciones.

13.28 Demuestre que si el problema de decisión del llenado de cajones se puede resolver en tiempo polinómico, el número óptimo de cajones se puede determinar en tiempo polinómico.

Sección 13.4 Algoritmos de aproximación

13.29 Podríamos plantear el problema de la satisfactibilidad como problema de optimización en esta forma:

Problema 13.16

Dada una fórmula CNF F, hallar una asignación de verdad para las variables de F que haga verdaderas el mayor número posible de cláusulas. ■

Describa el conjunto $FS(F)$ y el valor $val(F, x)$ para este problema (donde x es una solución factible).

13.30 Sea $F = \{S_1, \ldots, S_n\}$ un conjunto de subconjuntos de A tales que $\bigcup_{i=1}^{n} S_i = A$. Una *cobertura* de A es un subconjunto de F, digamos $\{S_{i_1}, \ldots, S_{i_k}\}$, tal que $\bigcup_{i=1}^{k} S_{i_j} = A$. ($F$ mismo es una cobertura de A.) Una *cobertura mínima* es una cobertura que usa el menor número posible de conjuntos. El problema de la *cobertura de conjunto* es:

Problema 13.17

Dado F según la descripción anterior, hallar una cobertura mínima de A. ■

Determine el conjunto $FS(F)$ y el valor $val(F, x)$ para este problema (donde x es una solución factible).

Sección 13.5 Llenado de cajones

13.31

a. Construya un ejemplo para el problema del llenado de cajones en el que el algoritmo FFD use tres cajones aunque el número óptimo sea dos.

b. Construya una sucesión infinita de ejemplos I_t, donde I_t tiene n_t objetos para algunos $n_1 < n_2 < \cdots < n_t$ y $opt(I_t) = 2$ pero FFD usa tres cajones.

13.32 Demuestre que el lema 13.10 no puede hacerse más categórico construyendo una sucesión de ejemplos tal que, para cada $k \geq 2$, haya una entrada con $opt(I)$ y FFD coloque $k - 1$ objetos en cajones extra.

13.33 Demuestre que, si $2 \leq opt(I) \leq 4$, FFD no usa más de $opt(I) + 1$ cajones.

13.34 Escriba un algoritmo de *mejor ajuste decreciente* para el llenado de cajones. ¿Qué orden tiene el tiempo de ejecución de peor caso?

13.35

a. Dé un ejemplo en el que la estrategia de *mejor ajuste decreciente* (*BFD*, por sus siglas en inglés) para llenar cajones produce un empacado que no es óptimo.

b. Dé un ejemplo en el que BFD produce un empaque distinto que el producido por FFD.

Sección 13.6 Los problemas de la mochila y de la sumatoria de subconjunto

13.36 Demuestre que la salida del algoritmo codicioso que describimos en el texto para el problema de la mochila simplificado (es decir, el problema de la sumatoria de subconjuntos) siempre es mayor que la mitad de la salida óptima. *Sugerencia:* Considere los dos casos: el resultado del algoritmo es mayor que $C/2$ y el resultado del algoritmo no es mayor que $C/2$.

13.37 Demuestre que si el algoritmo codicioso que describimos en el texto para el problema de la mochila simplificado (es decir, el problema de la sumatoria de subconjuntos) no considerara explícitamente el objeto de mayor tamaño, podría dar un resultado arbitrariamente alejado del óptimo. *Sugerencia:* Construya un ejemplo con sólo dos objetos.

⋆ **13.38** Idee un algoritmo que, dados n y k tales que $1 \leq k \leq n$, genere todos los subconjuntos de $N = \{1, 2, \ldots, n\}$ que contienen cuando más k elementos. El número de operaciones que se efectúan entre que se genera un subconjunto y se genera el siguiente deberá estar en $O(k)$ y ser independiente de n.

⋆ **13.39** Extienda los algoritmos de aproximación para el problema de la mochila simplificado y los teoremas 13.12 y 13.13 a la formulación general del problema (con utilidades además de tamaños).

⋆ **13.40** Al principio de la sección 13.6 se dijo que hay una sucesión de algoritmos $\mathbf{A}_k$ que se ejecutan en tiempo $O(n + k^2n)$ y hallan soluciones para el problema de la mochila simplificado que no difieren en más de un factor de $(1 + 1/k)$ del óptimo.

a. Para una entrada dada C, $(s_1, s_2, \ldots, s_n)$, explique cómo escoger k de modo que la solución producida por $\mathbf{A}_k$ sea óptima. *Sugerencia:* Recuerde que todas las cantidades de la entrada son enteras.

b. ¿El k escogido en la parte (a) da pie a un algoritmo que resuelve en tiempo polinómico el problema de la mochila simplificado? Explique por qué sí o por qué no.

Sección 13.7 Coloreado de grafos

13.41 Describa estructuras de datos para representar el grafo y el coloreado del algoritmo 13.3 de modo que la implementación sea rápida. ¿Qué complejidad tiene su implementación?

13.42 Demuestre el teorema 13.15.

13.43 Describa el comportamiento de la estrategia SCI con los grafos G_k definidos en la sección 13.7. En particular, ¿cuántas veces se intercambian pares de colores?

13.44 Suponga que $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, donde $|V_1| = n_1$, $|V_2| = n_2$, $|E_1| = m_1$ y $|E_2| = m_2$. ¿Cuántos vértices y aristas tiene $G = G_1[G_2]$? (Vea la definición 13.13.)

13.45 Demuestre que el grafo de la figura 13.14 es 3-coloreable.

13.46 Demuestre el lema 13.18.

13.47 Para $k = 3$, el lema 13.18 dice que si un grafo es 3-coloreable, el grafo de vecindad de cada vértice es 2-coloreable. El inverso dice: si la vecindad de todo vértice es 2-coloreable, el grafo es 3-coloreable. Si el inverso se cumple, existiría un algoritmo polinómico para el problema de la 3-coloreabilidad (pues es fácil determinar si todas las vecindades son 2-coloreables). Demuestre con un ejemplo que el inverso del lema 13.18 no se cumple.

13.48 Describa el coloreado que produciría el algoritmo de Wigderson (`color3`, algoritmo 13.4) con los grafos $G_k = (V_k, E_k)$, donde $V_k = \{a_i, b_i \mid 1 \leq i \leq k\}$ y $E_k = \{a_i b_j \mid i \neq j\}$. (En la sección 13.7.1 observamos que el algoritmo de coloreado secuencial podría funcionar de manera muy deficiente con estos grafos.)

Sección 13.8 El problema del vendedor viajero

13.49 Invente un ejemplo de grafo ponderado completo para el cual el recorrido hallado por la estrategia del vecino más cercano tiene menor peso que el recorrido hallado por la estrategia del eslabón más corto.

13.50 Invente un ejemplo de grafo ponderado completo para el cual la estrategia del vecino más cercano y la del eslabón más corto hallen recorridos óptimos.

13.51 Una extensión sencilla de la estrategia del vecino más cercano consiste en escoger una arista de peso mínimo que extienda *cualquiera* de los extremos del camino que se está construyendo (sin formar un ciclo).

a. Bosqueje el procedimiento de dicha extensión.

b. ¿Qué tan buena es la solución que halla para la figura 13.15, en comparación con la estrategia del vecino más cercano?

c. ¿Esa extensión siempre halla un recorrido por lo menos tan pequeño como, y posiblemente más pequeño que, el hallado por la estrategia del vecino más cercano? Justifique su respuesta con un argumento o un contraejemplo.

13.52 ¿Qué cambios hay que hacer a `eslabonMasCortoTSP` para procesar grafos dirigidos?

13.53 Considere el algoritmo de aproximación siguiente para el TSP. La estrada es un grafo G no dirigido, ponderado, completo, con n vértices y m aristas.

```
mstTSP(V, E W)
    Hallar un árbol abarcante mínimo para G; llámesele T.
    Escoger cualquier vértice v1 como raíz.
    Enumerar los vértices en el orden en que los visita un recorrido en orden previo de T;
    digamos, v1, ..., vn.
    Enviar a la salida el recorrido v1, ..., vn, v1.
```

a. Dé una buena cota superior para el tiempo de ejecución de peor caso de este algoritmo.

★ **b.** Demuestre que, si G satisface la desigualdad del triángulo, ecuación (13.8), el peso de un recorrido producido por este algoritmo no será mayor que dos veces el peso de un recorrido óptimo.

Sección 13.9 Computación por ADN

13.54 Demuestre que, si las cadenas R_i que representan los vértices en el algoritmo de Adleman se escogen arbitrariamente, es posible que en el paso 2c un hilo de ADN $\overline{R_i}$ que representa un vértice se una a un segmento complementario de un hilo de camino aunque el vértice v_i no aparezca en el camino representado por el hilo de camino. (Invente un ejemplo de cadenas para los vértices en el que suceda esto.)

Problemas adicionales

13.55 Para cada una de las afirmaciones siguientes, indique si es verdadera, falsa o desconocida. ("Desconocido" significa que todavía no se sabe si la afirmación es falsa o verdadera.) Piense bien sus respuestas.

a. El problema de la satisfactibilidad se puede reducir al problema de vendedor viajero.

b. Si $\mathcal{P} \neq \mathcal{NP}$, ningún problema en $\mathcal{NP}$ se puede resolver en tiempo polinómico.

c. 2-CNF (el problema de la satisfactibilidad cuando todas las cláusulas tienen exactamente dos literales) se puede reducir al problema de la satisfactibilidad.

d. No puede existir ningún algoritmo de aproximación (en tiempo polinómico) para colorear grafo que siempre use menos de $2\chi(G)$ colores para todos los grafos G, donde $\chi(G)$ es el número cromático de G, según la definición 13.1.

13.56 Considere el problema de optimización siguiente:

Problema 13.18

Dados $t_1, t_2, \ldots, t_n$, donde todos los t_i son enteros positivos, hallar una partición de esos enteros en dos subconjuntos de manera que la suma mayor sea mínima. ■

Éste podría verse como el problema de calendarizar trabajos en dos procesadores. El trabajo i tarda un tiempo t_i. Queremos terminar el conjunto de trabajos lo antes posible.

a. Escriba un algoritmo de aproximación **A** razonable, pero relativamente sencillo, que resuelva este problema en tiempo polinómico. (¿Cuánto tiempo tarda su algoritmo?)

b. Dé un ejemplo que muestre que su algoritmo no siempre produce un calendario óptimo.

★ **c.** Describa de la forma más amplia posible la calidad de las salidas de su algoritmo (o sea, las funciones S_A y R_A).

13.57 Considere esta generalización del problema anterior:

Problema 13.19

Se tienen p cajones con capacidad ilimitada y los enteros $t_1, \ldots, t_n$. Empacar los t_i en los cajones de modo que el nivel máximo de los cajones sea mínimo. ■

Piense en los cajones como procesadores y en los t_i como tiempos que tardan n trabajos independientes. El problema consiste en asignar trabajos a procesadores de modo que el conjunto de trabajos se termine en el menor tiempo posible.

Escriba un algoritmo de aproximación **A** para resolver este problema en tiempo polinómico. Describa lo más ampliamente que pueda la calidad de sus salidas.

13.58 Sea $G = (V, E)$ un grafo. Considere esta propuesta de algoritmo codicioso para hallar una cobertura de vértices mínima C para G. (Vea el problema 13.10, donde se definen las coberturas de vértices.)

```
Al principio ninguna arista está "marcada" y C está vacío;
while (queden aristas sin marcar)
    Escoger un vértice v en el que incida el mayor número de aristas no marcadas;
    Colocar v en C;
    Marcar todas las aristas que inciden en v.
```

Idee un ejemplo en el que este algoritmo no produzca una cobertura de vértices mínima.

⋆ **13.59** Suponga que tiene un subprograma, TSP, para resolver el problema de decisión del vendedor viajero en tiempo polinómico (es decir, dado un grafo ponderado completo y un entero k, determina si existe un recorrido cuyo peso total no rebase k).

a. Muestre cómo usaría el subprograma TSP para determinar el peso de un recorrido óptimo en tiempo polinómico.

b. Muestre cómo usaría el subprograma TSP para hallar un recorrido óptimo en tiempo polinómico.

⋆ **13.60** Demuestre que si existiera un algoritmo de aproximación de tiempo polinómico para el problema de la mochila que garantizara llenar la mochila con objetos cuyo valor total difiere del óptimo en una constante, se podría hallar una solución óptima en tiempo polinómico. (Dicho de otro modo, si existiera un algoritmo **A** que se ejecuta en tiempo polinómico y un entero k tal que, para todas las entradas I, $opt(I) - val(A, I) \leq k$, se podría hallar una solución óptima en tiempo polinómico.)

⋆ **13.61** Suponga que existe un algoritmo que resuelve el problema de la satisfactibilidad en tiempo polinómico, digamos `decisionSatisPoli(F)` que devuelve **`true`** si y sólo si la fórmula CNF F es satisfactible. Escriba un algoritmo de tiempo polinómico que, dada una fórmula CNF, halle una asignación de verdad para las variables que satisfaga la fórmula, si existe tal asignación, o indique que la fórmula no puede satisfacerse, en dado caso. Su algoritmo puede invocar `decisionSatisPoli` como subrutina.

Notas y referencias

Los dos trabajos que iniciaron el estudio intensivo de los problemas $\mathcal{NP}$-completos fueron Cook (1971) y Karp (1972). El segundo bosqueja demostraciones de reducibilidad entre muchos problemas $\mathcal{NP}$-completos. Tanto Stephen Cook como Richard Karp han ganado el Premio Turing de la ACM, y sus conferencias del Premio Turing (1983 y 1986, respectivamente) presentan perspectivas generales interesantes de la complejidad computacional y sus propias opiniones acerca del contexto de sus trabajos.

Una fuente importante de más pormenores, formalismos, aplicaciones, implicaciones, algoritmos de aproximación, etc., es Garey y Johnson (1979), un libro dedicado en su totalidad a los problemas $\mathcal{NP}$-completos. La definición de $\mathcal{NP}$ dada aquí utiliza una versión informal de la definición de máquinas de Turing no deterministas dada en Garey y Johnson. La obra también contiene una demostración del teorema de Cook, una demostración del teorema 13.22, una lista de varios centenares de problemas $\mathcal{NP}$-completos y una bibliografía larga (así que no repetiremos la mayor parte de las referencias originales aquí).

Los algoritmos de aproximación de las secciones 13.5 a 13.7 se tomaron de Sahni (1975) (mochila); Garey, Graham y Ullman (1972) y Johnson (1972) (llenado de cajones), y Johnson (1974) y Wigderson (1983) (coloreado de grafos). El esquema de aproximación más rápido que mencionamos para el problema de la mochila está en Ibarra y Kim (1975). Hay más algoritmos y referencias en Garey y Johnson. Bentley, Johnson, Leighton y McGeoch (1983) contiene estudios empíricos del comportamiento esperado de las heurísticas para llenado de cajones.

Hay algoritmos de aproximación para problemas de calendarización en Sahni (1976). Hochbaum (1997) es un libro acerca de algoritmos de aproximación para problemas $\mathcal{NP}$-completos. Lawler (1985) es un libro dedicado totalmente al problema del vendedor viajero. Garey y Johnson contiene más teoremas acerca de la baja probabilidad de hallar buenos algoritmos de aproximación para algunos problemas.

En Johnson y Trick (1996) se han reunido estudios empíricos de los problemas de camarilla, coloreado y satisfactibilidad.

El algoritmo de ADN de Adleman se describe en Adleman (1994, 1998). Kaplan, Cecchi y Libchaber (1995) intentaron repetir el experimento e informaron resultados "ambiguos". Hasta fines de 1998, no se sabía de otros intentos por repetir el experimento. La estimación de que un grafo de 70 vértices requeriría 10^{25} kilogramos es de Linial y Linial (1995). Un modelo de computación universal por ADN aparece en Kari, Păun, Rozenberg, Salomaa y Yu (1998); se han propuesto varios más. Păun, Rozenberg y Salomaa (1998) es un libro acerca de la computación por ADN. Maley (1998) reseña la computación por ADN incluyendo otros trabajos de laboratorio, y contiene una bibliografía extensa. Esta obra tiene una excelente introducción sin demasiados tecnicismos y explica muchos de los problemas y métodos.

14

Algoritmos paralelos

14.1 Introducción

Nuestro modelo de cómputo en casi todo este libro ha sido una computadora determinista, de acceso aleatorio y aplicación general que ejecuta una operación a la vez. Varias veces usamos modelos especializados para obtener cotas inferiores de varios problemas; ésos no eran máquinas de aplicación general, pero también ejecutaban una operación a la vez. Usaremos el término *algoritmo secuencial* para referirnos a los algoritmos acostumbrados paso por paso que hemos estado estudiando hasta ahora (y que también se conocen como *algoritmos seriales*). En este capítulo consideraremos *algoritmos paralelos*, algoritmos en los que es posible ejecutar varias operaciones al mismo tiempo en paralelo, es decir, algoritmos para máquinas que tienen más de un procesador trabajando en un problema a la vez.

En años recientes, al bajar el precio de los microprocesadores y mejorar la tecnología para interconectarlos, se ha vuelto posible y práctico construir computadoras paralelas de aplicación general que contienen un número muy grande de procesadores. El propósito de este capítulo es presentar algunos de los conceptos, modelos formales, técnicas y algoritmos del área del cómputo en paralelo.

Los algoritmos paralelos son naturales para muchas aplicaciones. En el procesamiento de imágenes (por ejemplo, en los sistemas de visión para robots) es posible procesar diferentes partes de una escena simultáneamente, es decir, en paralelo. El paralelismo puede acelerar el cómputo de pantallas gráficas. En los sistemas de búsqueda (por ejemplo, búsqueda bibliográfica, exploración de artículos noticiosos, edición de textos) es posible examinar en paralelo diferentes partes de la base de datos o texto. Los programas de simulación a menudo realizan el mismo cómputo para actualizar el estado de un gran número de componentes del sistema que se está simulando; esos cálculos se pueden efectuar en paralelo para cada paso de tiempo simulado. Las aplicaciones de inteligencia artificial (que incluyen procesamiento de imágenes y muchas búsquedas) se pueden beneficiar con la computación en paralelo. La transformada rápida de Fourier (sección 12.4) se implementa en hardware paralelo especializado. Algoritmos para resolver muchos problemas de optimización combinatoria (como las versiones de optimización de algunos de los problemas $\mathcal{NP}$-completos que describimos en el capítulo 13) implican examinar un gran número de soluciones factibles; una parte del trabajo se puede efectuar en paralelo. Los cálculos en paralelo también pueden acelerar considerablemente y sin dificultad la ejecución en otras áreas de aplicación.

En el caso de los ejemplos de aplicaciones en paralelo que acabamos de mencionar, y para algunos de los algoritmos que ya estudiamos en este libro, parece haber formas relativamente directas de dividir el cómputo en subcómputos paralelos. Muchos otros algoritmos muy conocidos y de amplio uso parecen ser inherentemente secuenciales. Por ello, se ha dedicado un gran esfuerzo tanto a hallar implementaciones paralelas de algoritmos secuenciales en los casos en que tal enfoque es fructífero, como a desarrollar técnicas totalmente nuevas para diseñar algoritmos paralelos.

14.2 Paralelismo, la PRAM y otros modelos

Si el número de procesadores de las computadoras paralelas fuera pequeño, digamos dos o seis, resultaría ventajoso en la práctica usarlos para resolver algunos problemas en los que el cálculo se aceleraría en un factor constante pequeño. Sin embargo, tales máquinas, y los algoritmos para ellas, no serían muy interesantes en el contexto de este libro, donde a menudo hacemos caso omiso de las constantes pequeñas. Los algoritmos paralelos se vuelven interesantes desde el punto de

vista de la complejidad computacional cuando el número de procesadores es muy grande, mayor que el tamaño de las entradas en muchos de los casos reales en los que se usa un programa. Es aquí donde podemos obtener aceleraciones importantes y algoritmos interesantes.

¿Qué tanto puede ayudarnos el paralelismo? Supóngase que un algoritmo secuencial para resolver un problema ejecuta $W(n)$ operaciones en el peor caso con una entrada de tamaño n y que tenemos una máquina con p procesadores. Entonces, lo mejor que podemos esperar de una implementación paralela del algoritmo es que se ejecute en tiempo $W(n)/p$, y no necesariamente podremos alcanzar esta aceleración máxima en todos los casos. Digamos que el problema consiste en ponerse calcetines y zapatos, y que un procesador es un par de manos. Un algoritmo secuencial común es: ponerse el calcetín derecho, ponerse el zapato derecho, ponerse el calcetín izquierdo, ponerse el zapato izquierdo. Si tenemos dos procesadores, podremos asignar uno a cada pie y llevar a cabo la tarea en dos unidades de tiempo en lugar de cuatro. Sin embargo, si tuviéramos cuatro procesadores no podríamos recortar el tiempo a una unidad, porque el calcetín tiene que ponerse antes que el zapato.

Hay varios modelos formales de aplicación general y aplicación especial de máquinas paralelas que corresponden a diversos diseños de hardware (reales o teóricos). Nos concentraremos en una clase importante de modelos para computadoras paralelas de aplicación general: la máquina paralela de acceso aleatorio (PRAM, por sus siglas en inglés). Aunque el modelo PRAM tiene algunas características poco realistas (que mencionaremos después), es una buena herramienta para introducir la computación en paralelo.

No siempre daremos el algoritmo más eficiente conocido para un problema; nuestro objetivo aquí es presentar algunas técnicas y algoritmos que se puedan entender sin gran dificultad. Puesto que el presente es un capítulo introductorio pequeño, se omitirá gran parte de lo interesante e importante del estudio de los algoritmos paralelos. En las Notas y referencias al final del capítulo se sugerirán unos cuantos temas adicionales y fuentes para los lectores que deseen investigar el tema.

14.2.1 La PRAM

Una *máquina paralela de acceso aleatorio* (*PRAM*, *parallel random access machine*) consiste en p procesadores de aplicación general $P_0, P_1, \ldots, P_{p-1}$, todos los cuales están conectados a una memoria de acceso aleatorio grande y compartida, M, que se trata como un arreglo (muy grande)

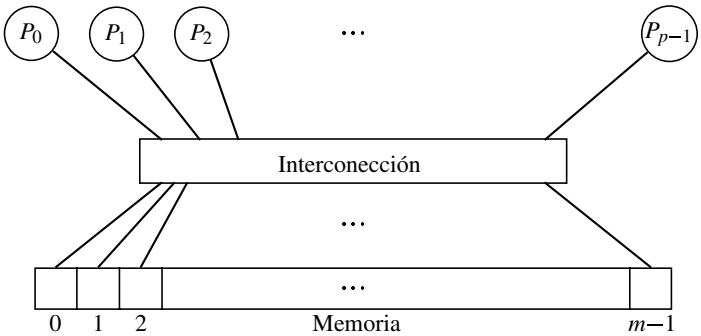


Figura 14.1 Una PRAM

de enteros (véase la figura 14.1). Los procesadores tienen una memoria privada (o local) para sus propios cálculos, pero toda la comunicación entre ellos se efectúa a través de la memoria compartida. A menos que se indique otra cosa, se supondrá que la entrada para un algoritmo está en las primeras n celdas de memoria, y que la salida se colocará en `M[0]` (o una sucesión inicial de celdas). También supondremos que todas las celdas de memoria que no contienen entradas contienen cero cuando inicia un programa de PRAM.

Todos los procesadores ejecutan el mismo programa, pero cada procesador "conoce" su propio índice (llamado *identificador de procesador* o *pid*) y "conoce" el tamaño de la entrada, que normalmente se designa con n, aunque a veces con un par (n, m) o algún otro conjunto fijo y pequeño de parámetros.[1] El programa podría ordenar a los procesadores hacer diferentes cosas dependiendo de sus pids. A menudo un procesador usa su pid para calcular el índice de la celda de memoria de la que debe leer o en la que debe escribir.

Los procesadores PRAM están sincronizados; es decir, todos inician cada paso al mismo tiempo, todos leen al mismo tiempo y todos escriben al mismo tiempo, dentro de cada paso. Algunos procesadores tal vez no lean o escriban en ciertos pasos. Cada paso de tiempo tiene dos fases: la fase de lectura, en la que cada procesador podría leer de una celda de memoria, y la fase de escritura, en la que cada procesador podría escribir en una celda de memoria. Cada fase podría incluir algún cálculo $O(1)$ empleando variables locales antes y después de su lectura o escritura. El tiempo que se concede para dichos cálculos es el mismo para todos los procesadores y todos los pasos, de modo que las lecturas y escrituras se mantengan sincronizadas. El modelo permite a los procesadores efectuar cálculos largos (pero en $O(1)$) en un paso porque, en los algoritmos paralelos, cabe esperar que la comunicación entre procesadores a través de la memoria compartida (es decir, leyendo y escribiendo) tome mucho más tiempo que las operaciones locales dentro de un procesador. Hay varios modelos con diferentes supuestos acerca de la cantidad de información que cabe en una celda de memoria y de las operaciones locales de las que se dispone. Los algoritmos que describimos en este capítulo funcionan con los supuestos más débiles, así que son robustos en este sentido.

En los modelos que consideraremos en este capítulo, cualquier cantidad de procesadores puede leer la misma celda de memoria de manera concurrente (es decir, en el mismo paso). Éste es el modelo de *lectura concurrente*. También hay modelos que prohíben las lecturas concurrentes: los modelos de *lectura exclusiva*. Hay varias otras variaciones de la PRAM dependiendo de la forma en que se manejan los conflictos de escritura. Después de examinar en la sección 14.3 unos cuantos algoritmos en los que los conflictos de escritura no constituyen un problema, consideraremos las variaciones en la sección 14.4.

Existen varios lenguajes de programación para describir algoritmos paralelos, pero usaremos una mezcla de español y nuestro lenguaje de pseudocódigo acostumbrado. En las cabeceras de procedimientos por lo regular se omiten los tipos porque las PRAM sólo usan enteros y arreglos, y los tipos quedan claros por el contexto. Varios de nuestros algoritmos tienen ciclos **`for`** y **`while`**. Cada procesador puede llevar la cuenta del índice del ciclo y efectuar los incrementos y pruebas apropiados durante las fases de cálculo de sus pasos.

Varios algoritmos utilizan arreglos almacenados en la memoria compartida. Podemos suponer que éstos se manejan igual que en los lenguajes de alto nivel; es decir, un compilador escoge cierta forma de acomodar los arreglos en la memoria después de su introducción, y traduce las re-

[1] Podríamos suponer que el tamaño de las entradas está en una posición de memoria global fija, pero eso simplemente añadiría una operación de lectura por tamaño de parámetro y no afecta el orden asintótico.

ferencias a arreglos a instrucciones que calculan direcciones de memoria específicas. Por ejemplo, si las entradas ocupan n celdas, y `alfa` es el tercer arreglo de k elementos, el compilador traduce una instrucción que pide al procesador P_i leer `alfa[j]` en instrucciones para calcular `indice = n + 2*k + j` y luego leer `M[indice]`. El cálculo de la dirección se lleva a cabo dentro de un paso de la PRAM.

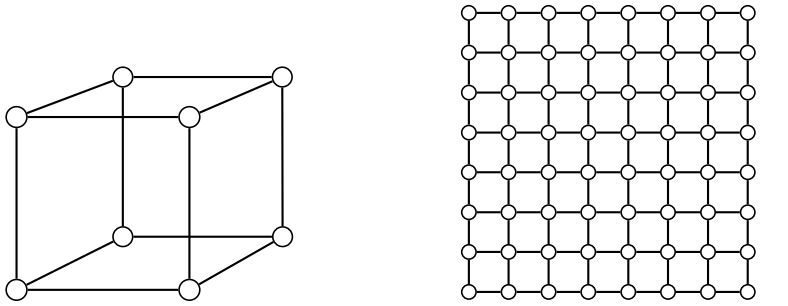
14.2.2 Otros modelos

Aunque la PRAM es un buen marco para desarrollar y analizar algoritmos que se ejecutan en máquinas paralelas, sería difícil o costoso crear el modelo en hardware real. La PRAM supone una red de comunicaciones compleja que permite a todos los procesadores acceder a cualquier celda de memoria simultáneamente, en un paso de tiempo, y escribir en cualquier celda en un paso de tiempo. Así, cualquier procesador se puede comunicar con cualquier otro en dos pasos: un procesador escribe datos en una posición de memoria en un paso y el otro lee esa celda en el siguiente paso. Otros modelos de computación en paralelo no tienen memoria compartida, lo que restringe la comunicación entre los procesadores. Un modelo que se parece más a hardware real es el *hipercubo*. Un hipercubo tiene 2^d procesadores, para alguna d (la *dimensión*), con cada uno conectado a sus vecinos. La figura 14.2(a) muestra un hipercubo de dimensión 3. Cada procesador tiene su propia memoria y se comunica con los demás procesadores enviando mensajes. En cada paso, cada procesador podría efectuar algo de cálculo y luego enviar un mensaje a uno de sus vecinos. Para comunicarse con un procesador no vecino, se puede enviar un mensaje con información de ruta incluida para indicar el destino final; el mensaje podría tardar hasta d pasos de tiempo en llegar a su destino. En un hipercubo con p procesadores, cada procesador está conectado a otros $\lg p$ procesadores.

Otra clase de modelos, llamados *redes de grado acotado*, restringe aún más las conexiones. En una red de grado acotado, cada procesador está conectado directamente a no más de otros g procesadores, para alguna constante g (el *grado*). Estas redes tienen diversos diseños; en la figura 14.2(b) se ilustra una red de 8×8. Los hipercubos y las redes de grado acotado son modelos más realistas que la PRAM, pero puede ser más difícil especificar y analizar los algoritmos para ellos. La selección de rutas para los mensajes entre procesadores, en sí un problema interesante, no existe en la PRAM.

Aunque la PRAM no es muy práctica, es fácil en lo conceptual trabajar con ella al desarrollar algoritmos. Por ello, se ha invertido mucho esfuerzo en hallar formas eficientes de simular cómputos de PRAM en otros modelos paralelos, sobre todo los que carecen de memoria compartida. Por ejemplo, cada paso de PRAM se puede simular en aproximadamente $O(\log p)$ pasos en una red de grado acotado. Así, podemos desarrollar algoritmos para la PRAM sabiendo que podremos traducirlos a algoritmos para máquinas reales. La traducción incluso podría ser automática empleando un programa traductor.

En el capítulo 13 definimos la clase de problemas $\mathcal{P}$ para facilitar la distinción entre problemas dóciles y renuentes. $\mathcal{P}$ consiste en problemas que se pueden resolver en tiempo polinómicamente acotado. También en el caso del cómputo en paralelo clasificamos los problemas según su uso de los recursos: tiempo y procesadores. La clase $\mathcal{NC}$ consiste en los problemas que se pueden resolver con un algoritmo paralelo, estando p (el número de procesadores) acotado por un polinomio en el tamaño de las entradas, y estando el número de pasos de tiempo acotado por un polinomio en el *logaritmo* del tamaño de las entradas. Dicho de forma más concisa, si el tamaño de las entradas es n, entonces $p(n) \in O(n^k)$ para alguna constante k, y el tiempo de peor caso, $T(n)$,



(a) Hipercubo (dimensión = 3) (b) Red de grado acotado (grado = 4)

Figura 14.2 Otras arquitecturas paralelas

está en $O(\log^m n)$ para alguna constante m. (Recordemos que $\lg^m n = (\lg n)^m$.) El origen del nombre $\mathcal{NC}$ se explica en las Notas y referencias al final del capítulo.

La cota de tiempo para $\mathcal{NC}$, que algunos llaman "tiempo poli-log" porque es un polinomio en términos del logaritmo de n, es muy pequeña; cabe esperar que los algoritmos paralelos se ejecuten muy rápidamente. La cota para el número de procesadores no es tan pequeña. Incluso con $k = 1$, podría no resultar práctico el uso de n^k procesadores si las entradas son moderadamente grandes. Los motivos para usar una cota polinómica, no un exponente específico, en la definición de $\mathcal{NC}$ son similares a los que tenemos para usar una cota polinómica en términos del tiempo para definir la clase $\mathcal{P}$. Una es que la clase de problemas que se pueden resolver en tiempo poli-log empleando un número polinómicamente acotado de procesadores es independiente del modelo de computación en paralelo específico que se use (de entre una clase amplia de modelos considerados "razonables"). Por tanto, $\mathcal{NC}$ es independiente de que usemos una PRAM o una red de grado acotado. Segunda, si un problema *no se puede* resolver rápidamente con un número polinómico de procesadores, ello indica claramente lo difícil que es el problema. De hecho, para casi todos los algoritmos que veremos el número de procesadores está en $O(n)$.

14.3 Algunos algoritmos de PRAM sencillos

En esta sección presentaremos algunas técnicas de uso común para el cómputo con PRAM y desarrollaremos algunos algoritmos sencillos que ilustran el "sabor" de los algoritmos para PRAM y al mismo tiempo proporcionan módulos o subrutinas que podremos usar más adelante.

En general, los algoritmos de PRAM son "teóricos" en el sentido de que demuestran que un problema se puede resolver en un tiempo que pertenece a una clase de orden asintótico específica. No existen PRAM reales que adquieran por arte de magia más procesadores si la entrada es más grande, sin límite. Por ello, no tiene mucho caso tratar de optimar los factores constantes, ya que el algoritmo nunca se ejecutará tal cual. En vez de ello, la presentación busca ser sencilla y clara.

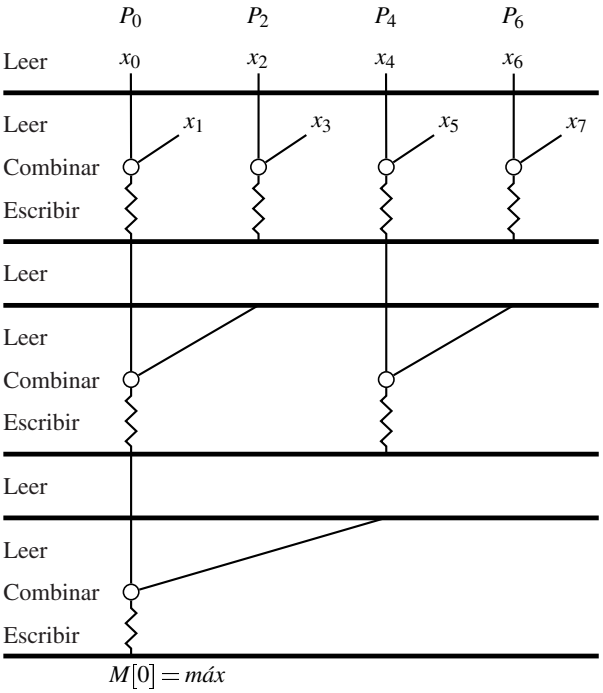


Figura 14.3 Torneo en paralelo: no se muestran los pasos de escritura en los ciclos en que ningún procesador escribe.

14.3.1 La técnica de abanico de entrada binario

Consideremos el problema de hallar la clave más grande en un arreglo de n claves. Hemos visto dos algoritmos para resolver este problema: el algoritmo 1.3 y el método de torneo descrito en la sección 5.3.2. En el algoritmo 1.3 examinamos sucesivamente los elementos del arreglo comparando `max`, la clave más grande hallada hasta el momento, con cada una de las claves restantes. Después de cada comparación, `max` podría cambiar; no podemos efectuar la siguiente comparación en paralelo porque no sabemos qué valor usar. Con el método de torneo, en cambio, se forman pares de elementos y se comparan en "rondas". En rondas sucesivas, se forman pares con los ganadores de la ronda anterior y se comparan (véase la figura 5.1). La clave más grande se encuentra en $\lceil \lg n \rceil$ rondas. Todas las comparaciones de una ronda se pueden efectuar simultáneamente. Por ello, el método de torneo nos ofrece naturalmente un algoritmo paralelo.

En un torneo, el número de claves que se consideran en cada ronda se reduce a la mitad, así que el número de procesadores requerido en cada ronda se reduce a la mitad. No obstante, con objeto de que la descripción del algoritmo sea corta y clara, especificaremos las mismas instrucciones para todos los procesadores en cada paso de tiempo. El trabajo extra efectuado podría causar confusión, por lo que recomendamos estudiar primero la figura 14.3, la cual muestra el trabajo que realmente contribuye a obtener la respuesta. Una línea recta representa una operación de *lectura*. Una línea en zig-zag representa una operación de *escritura*; un procesador escribe (la cla-

ve más grande que ha hallado) en la celda de memoria que tiene el mismo número que el procesador (o sea, P_i escribe en `M[i]`). Un círculo representa una operación binaria que "combina" dos valores; en este caso, se trata de una comparación que selecciona la mayor de dos claves. Los cálculos "contables" se acomodan en torno a las lecturas y escrituras. Si una línea *leer* incide en P_i proveniente de la columna de P_j, ello implica que P_i lee de `M[j]`, puesto que es ahí donde P_j escribió. La figura 14.4 muestra un ejemplo completo de la actividad de todos los procesadores. Las partes sombreadas corresponden a la figura 14.3 y muestran los cálculos que afectan a la respuesta.

Algoritmo 14.1 Torneo paralelo para el máximo

Entradas: Las claves `x[0]`, `x[1]`, …, `x[n−1]`, que inicialmente están en las celdas de memoria `M[0]`, `M[1]`, …, `M[n−1]`; y el entero *n*.

Salidas: La clave más grande estará en `M[0]`.

Comentarios: Cada procesador ejecuta el algoritmo utilizando su propio número de índice (`pid`) que especifica una distancia única a partir del principio de *M*. Se usa la variable `incr` para calcular el número superior de la celda a leer. Dado que *n* podría no ser una potencia de 2, el algoritmo inicializa las celdas `M[n]`, …, `M[2*n−1]` con $-\infty$ (algún valor pequeño) porque algunas de ellas participarán en el torneo.

```
torneoMaxParalelo(M, n)
   int incr;
1. Escribir −∞ (algún valor muy pequeño) en M[n+pid].
   incr = 1;
2. while (incr < n)
      Clave grande, temp0, temp1;
      Leer M[pid] y colocarlo en temp0.
3.    Leer M[pid+incr] y colocarlo en temp1.
      grande = max(temp0, temp1);
      Escribir grande en M[pid].
      incr = 2 * incr;
```

Es fácil analizar el algoritmo. La inicialización previa al ciclo **while** requiere un paso de lectura/escritura (paso 1), y cada iteración del ciclo **while** comprende dos pasos de lectura/escritura (pasos 2 y 3); el total es $2\lceil \lg n \rceil + 1$ pasos. Así pues, el algoritmo 14.1 utiliza *n* procesadores y tiempo $\Theta(\log n)$. (En realidad sólo necesita $n/2$ procesadores y un paso de lectura/escritura en el cuerpo del ciclo **while**, pero esto complica un poco las cosas; véase el ejercicio 14.3.)

El esquema de torneo, o abanico de entrada binario, empleado por el algoritmo 14.1 se puede aplicar también a varios otros problemas, por lo que vale la pena demostrar formalmente la corrección del algoritmo. Queremos demostrar (por inducción con *t*) que, después de la *t*-ésima iteración del ciclo **while**, `incr` $= 2^t$ y `M[i]` contiene el máximo de `x[i]`, …, `x[i+incr−1]`, con la convención de que `x[j]` $= -\infty$ si $j \geq n$. Así pues, cuando el ciclo termina después de $\lceil \lg n \rceil$ iteraciones, `M[0]` contendrá el máximo de `x[0]`, …, `x[n−1]`. Cabe señalar que estamos demostrando una afirmación más amplia que lo que en realidad nos interesa demostrar, para facilitar el uso de la inducción. Esto se conoce como fortalecimiento de la hipótesis inductiva.

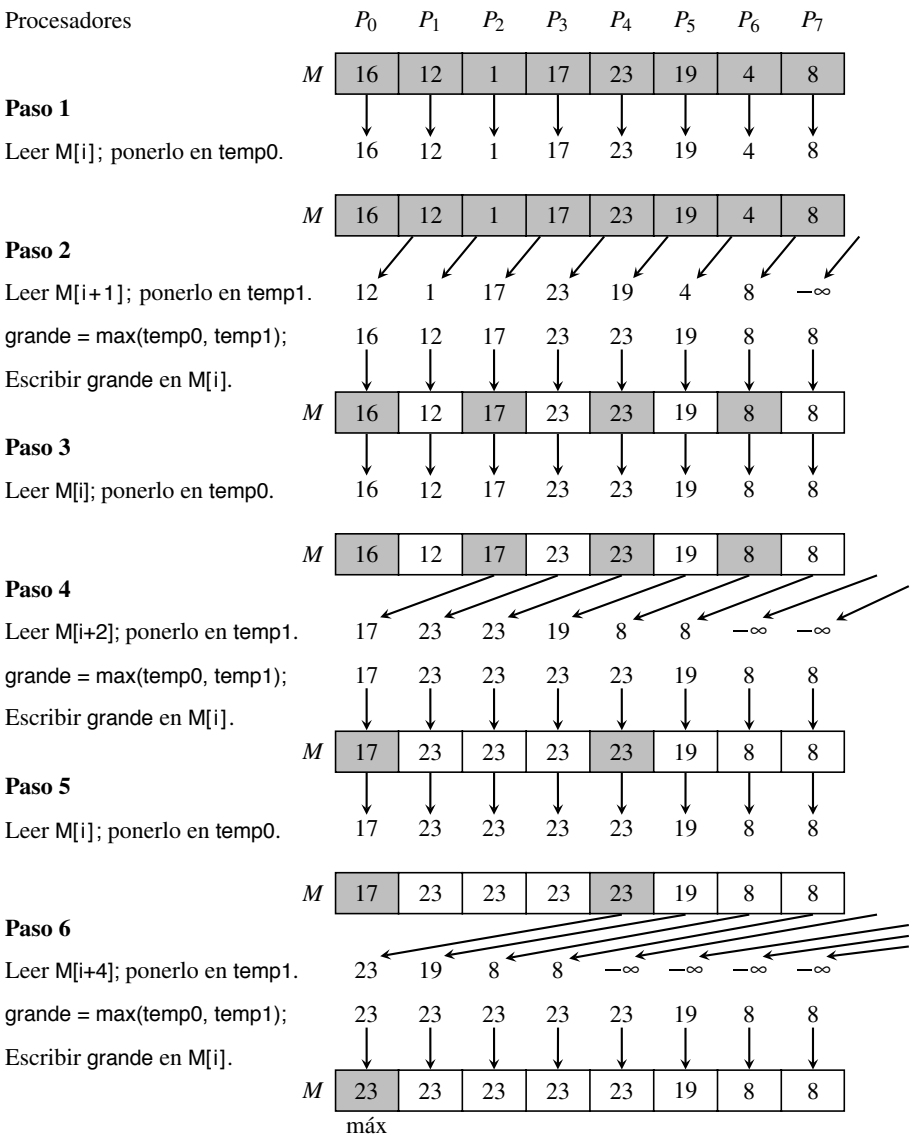


Figura 14.4 Ejemplo de torneo que muestra la actividad de todos los procesadores

Teorema 14.1 Al término de la t-ésima iteración del ciclo **while**, `incr` $= 2^t$ y cada celda `M[i]`, para $0 \leq i < 2^{\lceil \lg n \rceil}$, contiene el máximo de `x[i]`, ..., `x[i+incr-1]`. (Por tanto, si $t = \lceil \lg n \rceil$ e $i = 0$, se sigue la conclusión deseada.)

Demostración La demostración es por inducción con t, el número de iteraciones efectuadas. En toda la demostración, i es cualquier entero dentro del intervalo $0 \leq i < 2^{\lceil \lg n \rceil}$ a menos que se diga otra cosa. Como base de la inducción, sea $t = 0$. El teorema dice que, antes de ejecutarse el ciclo **while**, `M[i]` contiene el máximo de `x[i]`, ..., `x[i]` (o sea, `x[i]`), lo cual es verdad porque o bien ésa es la entrada o es el valor $-\infty$. También, por el paso de inicialización, `incr` $= 1 = 2^0$.

Sea ahora $t > 0$ y examinemos la t-ésima iteración del ciclo. Por la hipótesis inductiva, al término de la $(t - 1)$-ésima iteración, `incr` $= 2^{t-1}$, y `M[j]` contiene el máximo de `x[j]`, ..., `x[j+incr-1]`, para $0 \leq i < 2^{\lceil \lg n \rceil}$. Lo mismo se cumple al principio de la t-ésima iteración. Si $i \geq n$, `M[i]` no ha cambiado, y $-\infty$ es el máximo de todos los `x` cuyo índice es mayor que i porque todos son $-\infty$. Para $0 \leq i < p$, en la i-ésima iteración, los valores de las variables locales de P_i justo antes de la escritura son

$$\begin{aligned} \texttt{temp0} &= \text{máx}(\texttt{x[i]}, \ldots, \texttt{x[i+}2^{t-1} - 1\texttt{]}) \\ \texttt{temp1} &= \text{máx}(\texttt{x[i+}2^{t-1}\texttt{]}, \ldots, \texttt{x[i+}2^{t-1} + 2^{t-1} - 1\texttt{]}) \\ \texttt{grande} &= \text{máx}(\texttt{x[i]}, \ldots, \texttt{x[i+ }2^{t-1}\texttt{]}) \end{aligned}$$

Además, el nuevo valor de `incr` es 2^t. El valor de `grande` que se indica se escribe en `M[i]` durante el paso de escritura de la t-ésima iteración. Ello establece la afirmación de inducción para t y completa la demostración. □

Cabe señalar que el algoritmo 14.1 sobreescribe los datos de entrada. Si esto no es deseable, es cosa sencilla copiar las entradas (en un paso paralelo) en un área de "borrador" de la memoria y efectuar el cómputo ahí.

Basta una pequeña modificación al algoritmo 14.1 para poder usar el esquema de abanico de entrada binario para hallar la mínima de n claves, para calcular el *or* booleano o el *and* booleano de n bits, y para calcular la sumatoria de n claves, siempre en $\Theta(\log n)$ pasos, sin conflictos de escritura. El tema común de estos problemas es el uso de un operador binario asociativo para combinar todos los elementos de la entrada en un solo valor. La demostración de corrección también es válida, pues sólo usa el hecho de que la operación binaria sea asociativa.

14.3.2 Otros algoritmos fácilmente paralelizables

Es fácil hacer paralelos muchos algoritmos basados en arreglos o matrices porque es posible acceder simultáneamente a todas las partes de tales estructuras; no es preciso seguir "ligas" como en las listas ligadas y los árboles.

Problema 14.1 Multiplicación paralela de matrices

Consideremos el problema de multiplicar dos matrices A y B de $n \times n$. Recordemos la fórmula para obtener los elementos de la matriz producto C:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad \text{para } 0 \leq i, j < n.$$

Estamos usando índices que inician en 0 para las matrices. Desde luego, en este problema la salida no se coloca toda en `M[0]`; suponemos que se designan n^2 celdas de memoria para los elementos de C. ■

El algoritmo acostumbrado para multiplicar matrices tiene una versión paralela natural. Puesto que se permiten lecturas concurrentes, podemos asignar simplemente un procesador a cada elemento del producto, con lo que usaremos n^2 procesadores. Cabe señalar que n^2 es *lineal* en términos del tamaño de la entrada. Cada procesador P_{ij} puede calcular su c_{ij} en $2n$ pasos. (Hay que sumar n términos, y cada término requiere dos lecturas. Las multiplicaciones y sumas caben en estos pasos.) Si tenemos más procesadores podremos mejorar la rapidez.

Todas las multiplicaciones se pueden calcular y almacenar en dos pasos empleando n^3 procesadores. Es obvio que podemos usar el esquema de abanico de entrada binario del algoritmo 14.1 para sumar n enteros en $\Theta(\log n)$ pasos. El trabajo efectuado se podría representar como en la figura 14.3, sólo que los círculos representarían sumas en lugar de comparaciones. Por tanto, el producto de matrices se puede calcular en tiempo $O(\log n)$ con $\Theta(n^3)$ procesadores.

Problema 14.2 Cierre transitivo paralelo

Recordemos que el cierre transitivo (reflexivo) de una relación binaria A sobre un conjunto S es la relación binaria R (también sobre el conjunto S) que describe la alcanzabilidad en el grafo dirigido $G = (S, A)$ (definición 9.1). Es decir, $(u, v) \in R$ si y sólo si existe un camino de u a v en G. Se permiten caminos con cero aristas, así que $(v, v) \in R$ para todo $v \in S$, lo que implica que R es reflexivo. (A veces se define el cierre transitivo no reflexivo, en el que se exige que los caminos no estén vacíos.) La entrada es la representación de A como matriz de bits, donde 1 representa **`true`** y 0 representa **`false`**, con un bit por celda de memoria. La salida tiene el mismo formato. Al igual que en la multiplicación paralela de matrices, se designan n^2 posiciones de salida. ■

Este problema es un nivel más complejo que la multiplicación de matrices. Aunque el algoritmo 9.1, Cierre transitivo por atajos, no fue el algoritmo secuencial más eficiente, su estructura regular facilita la paralelización. Su ciclo **`while`** se ejecuta aproximadamente lg n veces en el peor caso. El *cuerpo* del ciclo **`while`** se puede paralelizar de forma muy similar a la multiplicación paralela de matrices (véase el ejercicio 14.4) y ejecutar en $O(\log n)$ pasos. Por tanto, el cierre transitivo se puede calcular en $O(\log^2 n)$ pasos sin conflictos de escritura.

Muchos algoritmos de programación dinámica se pueden acelerar fácilmente (aunque casi nunca a tiempo poli-log) efectuando cálculos en paralelo. Recordemos que las soluciones de programación dinámica por lo regular implican calcular elementos de una tabla. Es común que los elementos de una fila (o columna, o diagonal) sólo dependan de elementos de filas anteriores (o columnas, o diagonales). Por tanto, si hay n procesadores, se podrán calcular en paralelo todos los elementos de una fila de una tabla $n \times n$, recortando en un factor de n el tiempo de ejecución del algoritmo.

14.4 Manejo de conflictos de escritura

Los modelos de PRAM varían según la forma en que manejan los conflictos de escritura. El modelo de PRAM de Lectura Concurrente y Escritura Exclusiva (CREW, por sus siglas en inglés) exige que sólo un procesador escriba en una celda dada en cualquier paso dado; no se permiten algoritmos que pidan a más de un procesador escribir en una celda dada al mismo tiempo.

Hay varias formas de relajar la restricción CREW, las cuales reciben colectivamente el nombre de modelos CRCW (Lectura Concurrente, Escritura Concurrente).[2]

1. En el modelo de *Escritura Común* se permite a varios procesadores escribir en la misma celda al mismo tiempo si y sólo si todos escriben el mismo valor.
2. En el modelo de *Escritura Arbitraria*, si varios procesadores tratan de escribir en la misma celda de memoria al mismo tiempo, sólo lo logra uno de ellos, escogido arbitrariamente. Un algoritmo para este modelo debe funcionar correctamente sea cual sea el procesador que salga "victorioso" del conflicto de escritura.
3. En el modelo de *Escritura Prioritaria*, si varios procesadores intentan escribir en la misma celda de memoria al mismo tiempo, el que lo logrará será el que tenga el índice más pequeño.

Estos modelos CRCW son sucesivamente más categóricos, y todos son más categóricos que CREW: un algoritmo que sea válido y correcto para un modelo ubicado antes en la lista, será válido y correcto para todos los modelos posteriores, pero no viceversa.

Los modelos difieren en cuanto a la rapidez con que pueden resolver diversos problemas. Para ilustrar la diferencia, consideraremos el problema de calcular la función *or* booleana con n bits.

14.4.1 *Or* booleano con n bits

Utilizando el esquema de abanico de entrada binario del algoritmo 14.1, cada procesador ejecuta una operación *or* con un par de bits en cada ronda, y el problema se resuelve en tiempo $\Theta(\log n)$. Este método funciona con todos los modelos mencionados porque no hay conflictos de escritura; los procesadores escriben los resultados de sus operaciones en celdas de memoria distintas. ¿Podemos hallar un algoritmo aún más rápido?

Problema 14.3 *Or* booleano paralelo

Calcular el *or* de n bits $x_0, \ldots, x_{n-1}$, introducidos como ceros y unos en `M[0]`, ..., `M[n-1]`. ■

Se ha demostrado que la cota inferior de tiempo para el problema 14.3 en una PRAM CREW está $\Omega(\log n)$ (incluso si se usan más de n procesadores). En cambio, en todos los modelos CRCW el problema se puede resolver en tiempo constante.

Algoritmo 14.2 *Or* booleano con lectura común

Entradas: Los bits $x_0, \ldots, x_{n-1}$, en `M[0]`, ..., `M[n-1]`.

[2] *Advertencia:* Las abreviaturas empleadas para los diversos modelos en trabajos de investigación no son consistentes. EREW y CREW se usan de forma consistente para Lectura Exclusiva, Escritura Exclusiva y Lectura Concurrente, Escritura Exclusiva, respectivamente, pero CRCW podría referirse a cualquiera de varios modelos de escritura concurrente. Para evitar ambigüedades, escribiremos con palabras la regla para los conflictos de escritura.

Salidas: $x_0 \vee \cdots \vee x_{n-1}$ en `M[0]`.

```
orConEscrituraComun(M, n)
1. P_i lee x_i de [i];
   Si x_i es 1, entonces P_i escribe 1 en M[0].
```

Puesto que todos los procesadores que escriben en `M[0]` escriben el mismo valor, este programa es válido para el modelo de Escritura Común, y por ende también para los modelos de Escritura Arbitraria y Escritura Prioritaria. Así, el *or* de n bits se puede calcular en un paso en estos modelos con n procesadores. La técnica se puede aplicar al problema del cierre transitivo (véase el ejercicio 14.8).

14.4.2 Un algoritmo para hallar el máximo en tiempo constante

Si usamos el modelo PRAM de Escritura Común (o uno más categórico) podremos obtener un algoritmo para hallar el máximo de n números en menos tiempo que con el método de abanico de entrada binario. Este algoritmo emplea n^2 procesadores para simplificar la indización, aunque sólo efectúan trabajo $n(n-1)/2$ procesadores. La estrategia consiste en comparar todos los pares de claves en paralelo y luego comunicar los resultados a través de la memoria compartida. Utilizaremos un arreglo llamado `perdedor`. Recordemos que éste puede ocupar las celdas de memoria `M[n], ..., M[2*n-1]`, o algún otro segmento de la memoria global escogido por el compilador. En un principio, todos los elementos de este arreglo son cero (o se pueden inicializar con cero en un paso). Si x_i "pierde" en una comparación, se asignará el valor 1 a `perdedor[i]`.

Algoritmo 14.3 Máximo de n claves con escritura común

Entradas: Las claves $x_0, x_1, \ldots, x_{n-1}$, inicialmente en las celdas de memoria `M[0], M[1], ..., M[n-1]`; y el entero $n > 2$.

Salidas: La clave más grande estará en `M[0]`.

Comentarios: Por claridad, los procesadores se numerarán $P_{i,j}$, para $0 \leq i < j \leq n-1$. Cada procesador calculará i y j a partir de su índice (`pid`) con las fórmulas $i = \lfloor \texttt{pid}/n \rfloor$ y $j = \texttt{pid} - ni$. Si $i \geq j$, el procesador no efectúa trabajo. La figura 14.5 ilustra el algoritmo.

Procedimiento: Véase la figura 14.6. ■

Este algoritmo ejecuta sólo tres pasos de lectura/escritura. No obstante, el número de procesadores está en $\Theta(n^2)$. Si el número de procesadores se limita a n, la clave más grande se podrá hallar en tiempo $\Theta(\log \log n)$ con un algoritmo que aplica repetidamente el algoritmo 14.3 a grupos pequeños de claves. (Véanse Notas y referencias al final del capítulo.)

Este algoritmo pone de manifiesto que, si se permiten las escrituras comunes, el esquema de abanico de entrada binario no es la forma más rápida de hallar la clave máxima. En el ejemplo de multiplicación de matrices de la sección 14.3 sugerimos usar el abanico de entrada binario para suman n enteros en tiempo $\Theta(\log n)$. Quizá ahora el lector se pregunte si es posible sumar en tiempo constante con las PRAM de Escritura Común. En la sección 14.7 demostraremos que no lo es. Así pues, sumar n enteros es un problema más difícil que hallar el máximo de n enteros.

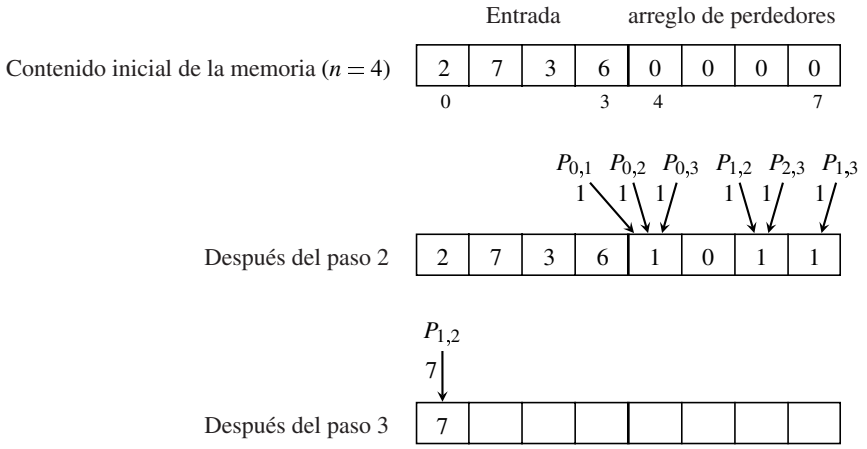


Figura 14.5 El algoritmo del máximo en tiempo constante

```
maxRapido(M, n)
```
2. Calcular i y j a partir de `pid`.
   `if (i >= j) return;`

   $P_{i,j}$ lee x_i (de `M[i]`).
2. $P_{i,j}$ lee x_j (de `M[j]`).
   $P_{i,j}$ compara x_i y x_j.
   Sea k el índice de la clave más pequeña (i si hay empate).
   $P_{i,j}$ escribe 1 en `perdedor[k]`.
   `// En este punto, todas las claves salvo la más grande`
   `// han perdido en una comparación.`
3. $P_{i,i+1}$ lee `perdedor[i]` (y $P_{0,n-1}$ lee `perdedor[n-1]`).
   El procesador que leyó un 0 escribe x_i en `M[0]`. ($P_{0,n-1}$ escribiría x_{n-1}.)
   `// Este procesador ya tiene en su memoria local el x que`
   `// necesita, por los pasos 1 y 2.`

Figura 14.6 Procedimiento para el algoritmo 14.3

14.5 Fusión y ordenamiento

No es difícil hallar formas de acelerar algunos de los algoritmos de ordenamiento del capítulo 4 ejecutando algunas de las operaciones en paralelo. El lector deberá poder escribir implementaciones paralelas de, por ejemplo, el Ordenamiento por Inserción o Mergesort capaces de ordenar n claves en tiempo $\Theta(n)$. En esta sección presentaremos un algoritmo de ordenamiento en paralelo basado en Mergesort que ejecuta aproximadamente $\lg^2(n)/2$ pasos de PRAM empleando n procesadores.

El algoritmo que presentamos aquí mejora drásticamente el ordenamiento secuencial (que está en $\Theta(n \log n)$). Por ejemplo, un arreglo de 1,000 claves se puede ordenar en 55 pasos paralelos; un algoritmo secuencial efectúa cerca de 10,000 comparaciones. No es éste el algoritmo de ordenamiento paralelo asintóticamente más rápido que se conoce; se puede ordenar en paralelo en tiempo $O(\log n)$ (en teoría; las constantes son demasiado grandes para que el método resulte práctico). El algoritmo que describiremos es muy fácil de entender, usa sólo n procesadores, además el número de pasos es un múltiplo pequeño de $\lg^2(n)$. Como siempre, supondremos que nos interesa ordenar en orden no decreciente.

14.5.1 Fusión en paralelo

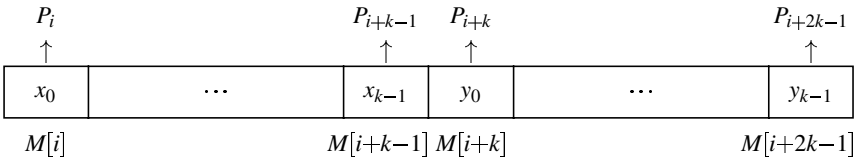
Como vimos en la sección 4.5, es posible fusionar dos sucesiones ordenadas, cada una de las cuales contiene $n/2$ claves, efectuando cuando más $n - 1$ comparaciones. El algoritmo de fusión que usamos ahí (algoritmo 4.4) parece básicamente secuencial: las dos claves que se comparan en un paso dependen del resultado de la comparación anterior. Aquí adoptaremos una estrategia distinta para fusionar en lg n pasos paralelos. Puesto que pensamos usar el algoritmo de fusión en un algoritmo al estilo Mergesort en el que siempre fusionaremos dos subintervalos de arreglo de la misma longitud, escribiremos el algoritmo de fusión para subintervalos del mismo tamaño. Es un ejercicio fácil generalizar el algoritmo y su análisis a subintervalos de tamaño distinto. La idea básica es la subrutina de *rango cruzado*.

Definición 14.1 Subrutina de rango cruzado

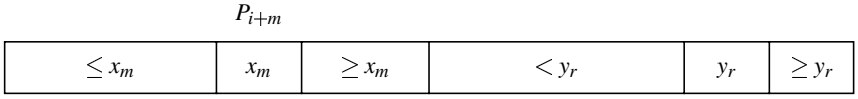
Dados dos arreglos ordenados, digamos X y Y, la *subrutina de rango cruzado* halla el rango en Y para cada elemento de X y el rango en X para cada elemento de Y. En términos específicos, el *rango cruzado* de $x \in X$ es el r más pequeño tal que $x \leq y_r$; es decir, si x se insertara en Y manteniendo el orden y colocando x en la posición más baja en caso de empates, x se colocaría en $Y[r]$ y su rango en Y sería r. Si x es mayor que todos los elementos de Y, su rango cruzado es uno más que el índice máximo de Y. También, el *rango cruzado* de $y \in Y$ es el r más pequeño tal que $y < x_r$. Obsérvese la asimetría de la definición, que hace que los elementos de X se traten como menores que los de Y en caso de un empate. ■

Supóngase que los dos arreglos ordenados están en las $2k$ celdas de memoria $M[i], \ldots, M[i + k - 1]$ y $M[i + k], \ldots, M[i + 2k - 1]$. Por claridad, nos referiremos al primer subintervalo como $X = (x_0, x_1, \ldots, x_{k-1})$ y al segundo como $Y = (y_0, y_1, \ldots, y_{k-1})$. Para implementar la determinación de rangos cruzados en paralelo, se asigna una clave a cada uno de los $2k$ procesadores, $P_i, \ldots, P_{i+2k-1}$, y se le encarga determinar el rango cruzado de esa clave. Un procesador al que se le asigna una clave de X, digamos x_m, efectúa una búsqueda binaria en Y para determinar el rango cruzado de x_m, llamémoslo $r(x_m)$. Asimismo, un procesador al que se le asignó una clave de Y, digamos y_m, determina el rango cruzado de Y_m, llamémoslo $r(y_m)$. Cada procesador recuerda el valor de r que calculó para el elemento que se le asignó.

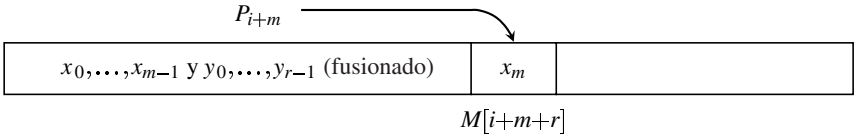
Ya estamos preparados para fusionar X y Y. Puesto que x_m está después de exactamente m claves en X y es mayor que $r(x_m)$ claves de Y, su posición correcta en el subintervalo fusionado es la celda $M[i + m + r(x_m)]$. De forma similar, y_m está después de exactamente m claves en Y y es mayor o igual que $r(y_m)$ claves de X, así que su posición correcta en el subintervalo fusionado es la celda $M[i + m + r(y_m)]$. (En el ejercicio 14.14 el lector demostrará que las posiciones de un elemento de X y un elemento de Y no pueden coincidir.) Una vez terminadas las búsquedas binarias, cada procesador escribe el elemento que se le asignó en la posición correcta. (Véase la figura 14.7, que ilustra el caso de x_m y el procesador P_{i+m}.)



(a) Asignación de procesadores a claves



(b) Pasos de búsqueda binaria: P_{i+m} halla r tal que $y_{r-1} < x_m \leq y_r$



(c) Paso de salida: P_{i+m} almacena x_m

Figura 14.7 Determinación de rangos cruzados y fusión en paralelo

Algoritmo 14.4 Fusión paralela

Entradas: Dos subintervalos de arreglo ordenados con k claves cada uno, en `M[i]`, ..., `M[i+k-1]` y `M[i+k]`, ..., `M[i+2*k-1]`.

Salidas: El arreglo fusionado, `M[i]`, ..., `M[i+2*k-1]`.

Comentarios: Participan los procesadores P_i, ..., P_{i+2k-1}. Cada procesador P_{i+m} tiene una variable local x (si $0 \leq m < k$) o y (si $k \leq m < 2k$) y otras variables locales para efectuar su búsqueda binaria. Cada procesador tiene una variable local `posicion` que indica dónde escribirá su clave.

Procedimiento: Véase la figura 14.8. ■

Teorema 14.2 El algoritmo de fusión paralela ejecuta $\lfloor \lg k \rfloor + 2$ pasos para fusionar dos subintervalos de arreglo ordenados, cada uno de los cuales tiene k claves, utilizando $2k$ procesadores.

Demostración La inicialización es un paso de PRAM. Todas las búsquedas binarias se efectúan en subintervalos de k claves, así que requieren $\lfloor \lg k \rfloor + 1$ pasos de lectura/cálculo. Puesto que durante las búsquedas binarias no se escribe en la memoria compartida, la salida puede generarse en el último paso de la búsqueda binaria. Por tanto, el total es $\lfloor \lg k \rfloor + 2$. □

Cabe señalar que, dado que no hay conflictos de escritura, el algoritmo de fusión funciona con todas las variaciones de la PRAM que hemos descrito.

```
fusionParalela(M, i, k)
    int r, posicion;
    TipoClave x, y;
    // Inicialización:
    Si m < k, P_{i+m} lee M[i+m] y lo coloca en x.
    Si m ≥ k, P_{i+m} lee M[i+m] y lo coloca en y.

    // Pasos para determinar rangos cruzados:
    Los procesadores P_{i+m}, para 0 ≤ m < k, determinan el rango cruzado de x en
    M[i+k], ..., M[i+2*k-1] y guardan el resultado localmente en r.
    (Simultáneamente) los procesadores P_{i+m}, para k ≤ m < 2k, determinan el rango cru-
    zado de y en M[i], ..., M[i+k-1] y guardan el resultado localmente en r.

    // Paso de salida:
    Cada P_{i+m} (para 0 ≤ m < k) calcula posicion = i + m + r.
    Cada P_{i+k+m} (para 0 ≤ m < k) calcula posicion = i + m + r.
    Cada P_{i+m} (para 0 ≤ m < 2k) escribe su clave (x o y) en M[posicion].
```

Figura 14.8 Procedimiento para el algoritmo 14.4

14.5.2 Ordenamiento

Supóngase que tenemos un arreglo de n claves a ordenar. Recordemos la estrategia de Mergesort:

> Dividir el arreglo en dos mitades.
> Ordenar las dos mitades (recursivamente).
> Fusionar las dos mitades ordenadas.

Si "desenrollamos" la recursión, veremos que el algoritmo primero fusiona pequeños subintervalos ordenados del arreglo (de una clave cada uno), luego fusiona subintervalos un poco mayores (de dos claves cada uno), luego subintervalos más grandes y así hasta fusionar dos subintervalos de tamaño (aproximado) $n/2$. El algoritmo recursivo fusiona subintervalos más grandes antes de procesar todos los subintervalos pequeños (porque ordena totalmente la primera mitad de las claves antes de comenzar a trabajar con la segunda mitad). Si queremos escribir un algoritmo paralelo iterativo sistemático, fusionaremos todos los pares de subintervalos de tamaño 1 en la primera pasada (en paralelo), luego fusionaremos todos los pares de subintervalos de tamaño 2 en la siguiente pasada, y así sucesivamente. Es obvio que efectuaremos $\lceil \lg n \rceil$ pasadas de fusión. La asignación de procesadores a sus tareas es muy fácil. Para fusionar dos subintervalos de arreglo que ocupan, digamos, `M[i]`, ..., `M[j]`, los procesadores P_i, ..., P_j realizan la fusión empleando el algoritmo 14.4. La figura 14.9 ilustra una pasada.

Algoritmo 14.5 Ordenamiento por fusión

Entradas: Un arreglo de n claves en `M[0]`, ..., `M[n-1]`.

Salidas: Las n claves ordenadas en orden no decreciente, en `M[0]`, ..., `M[n-1]`.

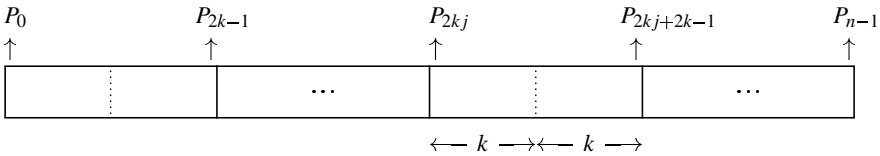


Figura 14.9 Asignación de procesadores para una pasada de fusión: los procesadores $P_{2kj}, \ldots, P_{2kj+2k-1}$ fusionan el j-ésimo par de subintervalos de arreglo de tamaño k.

Comentarios: La indización en el algoritmo se facilita si el número de claves es una potencia de 2, así que el primer paso "rellena" la entrada con claves grandes al final. De todos modos se usan sólo n procesadores.

```
mergeSortParalelo(M, n)
    int k;   // el tamaño de los subintervalos a fusionar
    P_i escribe ∞ (alguna clave grande) en M[n+i].
    for (k = 1; k < n; k = 2 * k)
        Para cada i = 0, 2k, 4k, 6k, ..., i < n (simultáneamente)
            P_i, ..., P_{i+2k-1} ejecutan fusionParalela(M, i, k).
```

Teorema 14.3 El algoritmo 14.5 ordena n claves en $(\lceil \lg n \rceil + 1)(\lceil \lg n \rceil + 2)/2$ pasos. Por tanto, el ordenamiento en paralelo se puede efectuar en tiempo $O(\log^2 n)$ con n procesadores.

Demostración En la t-ésima pasada por el ciclo **for**, cada uno de los subintervalos que se están fusionando tiene $k = 2^{t-1}$ claves así que, por el teorema 14.2, la t-ésima ejecución de `fusionParalela` ejecuta $t + 1$ pasos. Se efectúan $\lceil \lg n \rceil$ pasadas porque k aumenta al doble después de cada pasada. En total, todas las pasadas ejecutan

$$\sum_{t=1}^{\lceil \lg n \rceil} (t + 1) = \tfrac{1}{2} (\lceil \lg n \rceil + 1) (\lceil \lg n \rceil + 2) - 2$$

pasos, y además hay un paso de inicialización. □

14.6 Determinación de componentes conectados

En el capítulo 7 estudiamos un algoritmo secuencial, el algoritmo 7.2, para determinar los componentes conectados de un grafo no dirigido (o de un digrafo simétrico) G. Se usó búsqueda primero en profundidad y el tiempo de ejecución estaba en $\Theta(n + m)$. Aunque la búsqueda primero en profundidad podría parecer inherentemente secuencial, existen algoritmos paralelos rápidos para construir árboles de búsqueda primero en profundidad. Sin embargo, no es necesario efectuar una búsqueda primero en profundidad para determinar componentes conectados. (Por ejemplo, se puede usar búsqueda primero en amplitud.) ¿Qué tanto podemos acelerar la resolución con más procesadores?

En esta sección, $G = (V, E)$ es un grafo no dirigido con $|V| = n$ y $|E| = m$. (Como digrafo simétrico, $|E| = 2m$.) Para no complicar la notación, sea $V = \{1, 2, \ldots, n\}$. El grafo se presenta en la entrada como los dos parámetros de tamaño, n y m, y una sucesión de $2m$ enteros que representan las m aristas.

Es relativamente sencillo determinar componentes conectados en tiempo $O(\log n)$ utilizando n^3 procesadores en el modelo de Escritura Común. La idea consiste en determinar primero el cierre transitivo (véase el ejercicio 14.8) y luego, en paralelo para cada vértice v, hallar un identificador para el componente conectado de v. Los detalles se dejan para el ejercicio 14.18. Puesto que un grafo no tiene más de $O(n^2)$ aristas, el número de procesadores empleado crece a un ritmo más que lineal al crecer el tamaño de las entradas. Sin embargo, el cierre transitivo de G contiene mucha más información de la que necesitamos. ¿Podemos resolver el problema de los componentes conectados con un algoritmo que use un número lineal de procesadores?

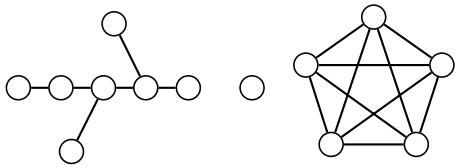
En esta sección presentaremos un algoritmo paralelo que halla componentes conectados en tiempo $O(\log n)$ utilizando máx$(n + 1, 2m)$ procesadores. El algoritmo tiene conflictos de escritura genuinos; no sólo cabe la posibilidad de que varios procesadores intenten escribir en la misma celda; también podrían estar tratando de escribir valores distintos. De las variaciones de la PRAM que hemos descrito, no podemos usar los modelos CREW ni de Escritura Común. Aquí es preciso usar la PRAM de Escritura Prioritaria o la PRAM de Escritura Arbitraria, que es menos categórica. Demostraremos la corrección en el modelo menos categórico, pues de ella se sigue la corrección en el modelo más categórico.

14.6.1 Estrategia y técnicas

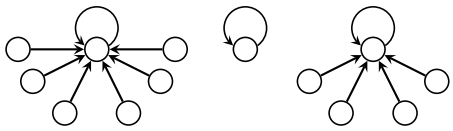
El algoritmo de componentes conectados es más complicado que cualquiera de los otros algoritmos paralelos que hemos visto hasta ahora. Presentaremos una descripción de alto nivel del algoritmo, y luego mostraremos cómo pueden implementarse sus diferentes partes en una PRAM. Presentemos algo de terminología.

Definición 14.2 Supervértice, estrella

Dado un bosque de árboles adentro (en el que las aristas apuntan de los vértices hacia sus padres, como en la sección 2.3.5), un *supervértice* es el conjunto de vértices de cualquier árbol individual y una *estrella* es un árbol de altura 0 o 1. En la figura 14.10 se muestra un ejemplo. ■



(a) Un grafo



(b) Sus componentes como estrellas

Figura 14.10 Componentes conectados convertidos en estrellas: las aristas que van de una raíz a sí misma se incluyen por comodidad de contabilización.

El algoritmo inicia con cada vértice en un árbol adentro aparte y luego combina repetidamente árboles que pertenecen al mismo componente conectado, formando supervértices más grandes, y acorta los árboles. Al final, cada componente conectado se convierte en una estrella. Los árboles adentro se representan con un arreglo `padre`, tal que `padre[v]` es el padre del vértice v. Por convención, el padre de una raíz es la raíz misma. Una vez que los componentes conectados se hayan convertido en estrellas, podremos determinar en tiempo constante si dos vértices están en el mismo componente comparando sus padres.

Los lectores que hayan estudiado relaciones de equivalencia dinámica y programas Unión-Hallar en la sección 6.6 notarán muchas similitudes con ellos en la descripción anterior. Efectivamente, los componentes conectados definen una relación de equivalencia sobre un grafo no dirigido. Recordemos que `hallarC` (`hallar` con compresión de caminos) también acortaba árboles adentro, que se formaban mediante operaciones `union`. Tener presentes tales similitudes es de utilidad al ahondar en los pormenores del método paralelo.

El algoritmo usa de forma repetida dos técnicas básicas: *atajos* y *enganchado*. Los atajos acortan árboles, y también son útiles en otros algoritmos paralelos para grafos. Resulta interesante comparar esta operación con la *compresión de caminos*.

Definición 14.3 Atajos

El uso de *atajos* (también llamado *doblado* o *salto de apuntadores*) simplemente cambia el padre de un vértice v al abuelo actual de v:

```
padre[v] = padre[padre[v]].
```
■

Los atajos se aplican en paralelo a todos los vértices. Para entender con qué rapidez esta operación puede recortar caminos largos, consideremos una cadena sencilla de vértices como la de la figura 14.11(a), donde `padre[v]` $= v - 1$ (y `padre[1]` $= 1$). Las partes (b) y (c) de la figura 14.11 muestran los apuntadores de `padre` después de la primera y segunda aplicaciones de la operación de atajos. Si inicialmente hay n vértices en la cadena, después de $\lceil \lg n \rceil$ aplicaciones de los atajos todos los vértices tendrán el mismo padre.

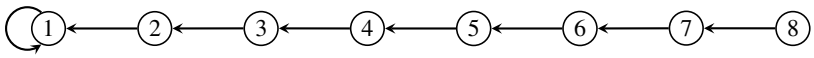
Recordemos que el grafo $G = (V, E)$ en el que el algoritmo está determinando componentes conectados no es dirigido, pero el bosque de árboles adentro que el algoritmo manipula sí es dirigido y sus aristas son diferentes de las de G. Necesitamos saber si estamos hablando de una arista (no dirigida) de G o de una arista dirigida del bosque. El término *padre* sólo tiene sentido en el bosque y las aristas dirigidas del bosque son (v, `padre[v]`), siempre que `padre[v]` sea distinto de v.

Los atajos nunca unen dos árboles distintos. Necesitamos la operación de enganchado para conectar árboles. Esto es análogo a la operación `union` de un programa Unión-Hallar.

Definición 14.4 Enganchado

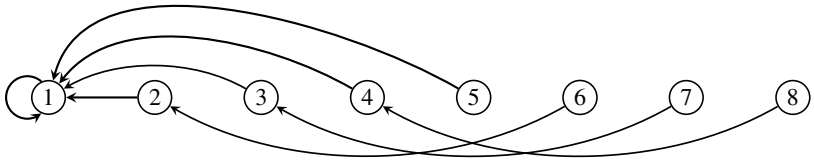
La operación `enganchar(i, j)` conecta la raíz del árbol adentro de i al padre de j como hijo nuevo. Decimos que el árbol adentro de i está *enganchado al* padre de j. (Cabe señalar que tanto i como j podrían ser su propio "padre" en el arreglo `padre`.) El algoritmo sólo aplica `enganchar(i, j)` cuando `padre[i]` es una raíz (es decir, i es una raíz o un hijo de una raíz). Así pues, la operación se puede implementar con

```
padre[padre[i]] = padre[j].
```



(a) Una cadena de vértices

(b) Después de una aplicación de atajos



(c) Después de dos aplicaciones de atajos

Figura 14.11 Efecto de aplicar atajos en un ejemplo sencillo: después de una tercera operación de atajos, todos los vértices apuntarán a la raíz.

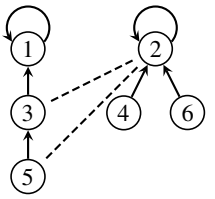
El algoritmo emplea ciertos casos especiales de enganchado:

1. *Enganchado de estrella condicional*:
   Si *i* está en una estrella, *j* está adyacente a *i* en *G*, y el padre de *j* es menor que el padre de *i*, entonces `enganchar(i, j)`. El requisito de conectarse al menor de los dos padres ayuda a evitar la introducción de ciclos. El enganchado de estrella condicional se ilustra en la figura 14.12, partes (a) y (b).
2. *Enganchado de estrella incondicional*:
   Si *i* está en una estrella, *j* está adyacente a *i* en *G*, y *j* no está en la estrella de *i*, entonces `enganchar(i, j)`. El enganchado de estrella incondicional se ilustra en la figura 14.12, partes (c) y (d).

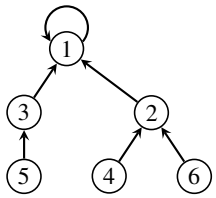
Cabe señalar que el algoritmo exige que *i* esté en una estrella en ambos casos. ■

En cualquier momento, podría haber varios pares de vértices, *i* y *j*, que satisfacen las condiciones para engancharse, pero sólo un valor se puede almacenar como nuevo padre de la raíz de *i*. En el algoritmo paralelo, diferentes procesadores estarán probando las diferentes opciones, y varios procesadores podrían tratar de escribir en `padre[padre[i]]` al mismo tiempo. Por ejemplo, en las partes (c) y (d) de la figura 14.12 mostramos el resultado de `enganchar(8, 10)`, que modifica `padre[7]`. Los requisitos de las operaciones `enganchar(7, 11)` y `enganchar(8, 11)` se satisfacen en la figura 14.12(c), así que otros procesadores las ejecutarán, y también tratarán de escribir en `padre[7]`. Sólo un procesador logrará escribir, pero el algoritmo funcionará correctamente sea cual sea el que lo logre.

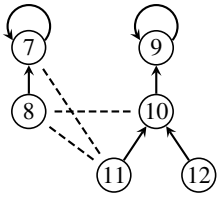
Cabe señalar que dos árboles se enganchan sólo si existe una arista de *G* que incide en un vértice de cada árbol. Por tanto, un supervértice siempre es un subconjunto de un componente co-



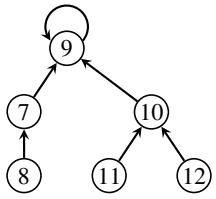
(a) Árboles antes del enganchado: las aristas punteadas están en G.



(b) Enganchado de estrella condicional: `enganchar(2, 3)`.



(c) Árboles antes del enganchado: las aristas punteadas están en G.



(d) Enganchado de estrella incondicional: `enganchar(8, 10)`.

Figura 14.12 Ilustraciones de enganchado

nectado. Si se ejecuta durante suficiente tiempo, el algoritmo tarde o temprano engancha todos los árboles que forman parte de un componente conectado.

14.6.2 El algoritmo

El algoritmo inicia con cada vértice de G en un árbol aparte, así que cada vértice es en sí una estrella, al principio. El algoritmo efectúa enganchado y aplica atajos una y otra vez hasta lograr la estructura deseada. Primero daremos una descripción de alto nivel.

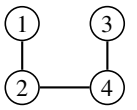
Con base en nuestra experiencia con los programas Unión-Hallar en la sección 6.6, cabría esperar que la inicialización de cada vértice de modo que sea su propio árbol (es decir, `padre[v] = v`) bastaría para poner en marcha el algoritmo. Lamentablemente, eso no funciona realmente. Después de presentar el algoritmo, explicaremos el problema y la solución.

Algoritmo 14.6 Componentes conectados en paralelo (bosquejo)

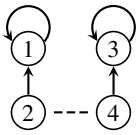
Entradas: Un grafo no dirigido $G = (V, E)$.

Salidas: Un bosque de árboles dirigidos con altura máxima de 1, representado por el arreglo `padre`, cuyos índices son los vértices. Cada árbol contiene los vértices de un componente conectado.

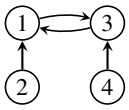
Comentarios: Una instrucción especificada para un vértice v se ejecuta en paralelo para todos los vértices. Los pasos de enganchado se ejecutan en paralelo para todas las aristas ij de G (y *sólo* para los pares i y j tales que ij sea una arista). Cada arista, digamos xy, se procesa dos veces (en



(a) El grafo *G*



(b) Después del enganchado estrella condicional



(c) Después del enganchado estrella incondicional: Ambos `enganchar (2, 4)` y `enganchar (4, 2)`

Figura 14.13 Introducción de un ciclo durante el primer paso y después de la inicialización defectuosa

paralelo); en la primera *x* hace las veces de *i* y en la otra *y* hace las veces de *i*. La subrutina `inicCCParalelo` se describe en el texto.

```
CompsConecParalelo(G, n, m)  // BOSQUEJO
    inicCCParalelo(G, n, m);
    Mientras la aplicación de atajos produzca cambios:
        // Enganchado de estrella condicional
           Si ij ∈ E, i está en una estrella y padre[i] > padre[j]:
              enganchar(i, j):
           // Enganchado de estrella incondicional
           Si ij ∈ E, i está en una estrella y padre[i] ≠ padre[j]:
              enganchar(i, j);
           // Atajos
           Si v no está en una estrella:
              padre[v] = padre[padre[v]];
```

Un hecho que se usa para demostrar que el algoritmo funciona correctamente es que los enganchados de estrella condicional e incondicional no producen estrellas nuevas porque el nuevo supervértice forma un árbol cuya altura es 2 o más. El problema es que sí podrían hacerlo durante la primera pasada por el ciclo si `inicCCParalelo` no hiciera más que convertir a cada vértice en su propia estrella. Los vértices individuales (árboles con altura 0) podrían formar un árbol de altura 1 (una estrella) al engancharse. En tal caso, el paso de enganchado de estrella incondicional podría enganchar dos estrellas en ambas direcciones y crear así un ciclo. En la figura 14.13 se ilustra este caso. El problema se elimina si `inicCCParalelo` se asegura de que todos los solitarios (árboles de un solo vértice) se enganchen a algo o que algo se enganche a ellos (a menos que el vértice esté aislado, es decir, no participe en ninguna arista de *E*). He aquí la inicialización correcta.

Algoritmo 14.7 Inicializar para componentes conectados

Entradas: Las mismas del algoritmo 14.6.

Salidas: El arreglo `padre` representa un bosque de árboles adentro, todos los cuales tienen altura 1, salvo los nodos aislados. Es decir, todo vértice que incida en cualquier arista de G estará en un árbol de altura 1. Además, todas las aristas del bosque unirán dos vértices que en G están conectados (mediante un camino de longitud 1 o 2).

```
inicCCParalelo(G, n, m)
    Calcular v, i y j a partir de pid.
    padre[v] = v;
    if (ij ∈ E && i > j) // Enganchado condicional de solitarios
        enganchar(i, j);
    if (ij ∈ E && i es un solitario)  // Enganchado incondicional de solitarios
        enganchar(i, j);
```

La figura 14.14 ilustra la acción del algoritmo. La corrección del algoritmo se basa en dos teoremas que a su vez se demuestran en una serie de lemas. El algoritmo en sí no es muy difícil de entender si se ejecutan a mano unos cuantos ejemplos, por lo que recomendamos examinar detenidamente la figura 14.14 antes de continuar. (Obsérvese cómo el algoritmo 14.7 protege los árboles con raíz en 5 y 7 del problema ilustrado en la figura 14.13.)

Teorema 14.4 En cualquier momento durante la ejecución del algoritmo 14.6, la estructura definida por los apuntadores `padre` es un bosque.

Teorema 14.5 Cuando el algoritmo 14.6 termina, el bosque definido por los apuntadores de `padre` consta únicamente de estrellas, y los vértices de cada estrella son exactamente los vértices de un componente conectado de G.

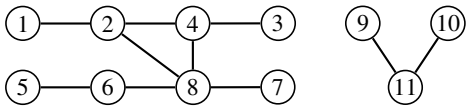
Las demostraciones de los teoremas utilizan los lemas siguientes.

Lema 14.6 Después de la inicialización, la estructura definida por los apuntadores de `padre` es un bosque. Todos los árboles tienen al menos dos vértices, con excepción de los árboles que consisten en un vértice que en G está aislado (es decir, que es un componente conectado de G).

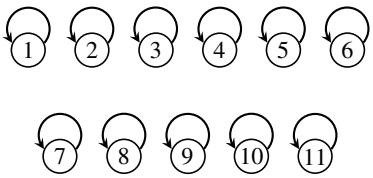
Demostración Ejercicio 14.22. □

Lema 14.7 Los enganchados de estrella condicional e incondicional nunca crean estrellas nuevas; es decir, el supervértice resultante forma un árbol con una altura de por lo menos 2.

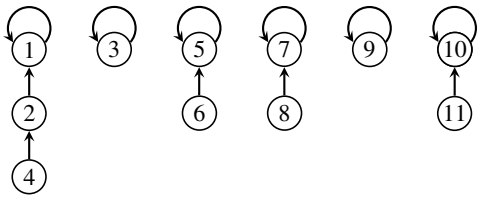
Demostración Los solitarios que sigan existiendo después de la inicialización nunca se engancharán a ninguna otra cosa. Si la raíz de un árbol que tiene por lo menos dos nodos se engancha a otro árbol, el nuevo árbol tendrá una altura de por lo menos 2. □



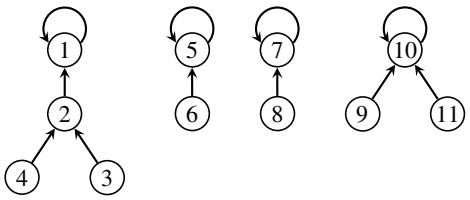
(a) El grafo



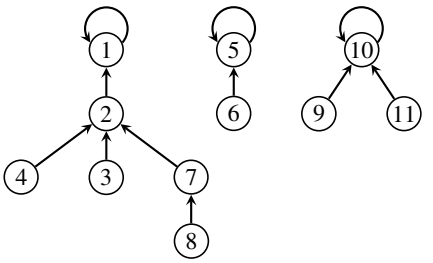
(b) El bosque inicial



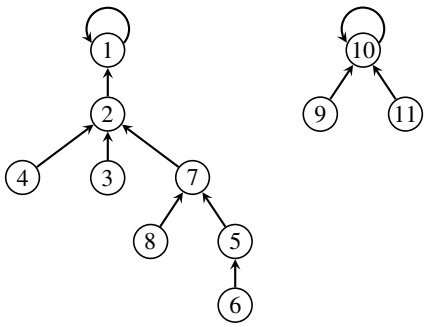
(c) Después del enganchado condicional de solitarios: $P_{4,2}$, $P_{2,1}$, $P_{6,5}$, $P_{8,7}$ y $P_{11,10}$ lograron escribir



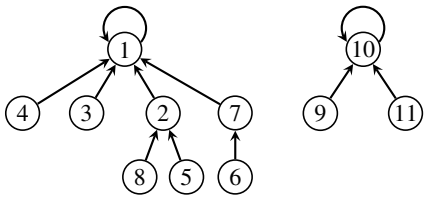
(d) Después del enganchado incondicional de solitarios: $P_{3,4}$ y $P_{9,11}$ lograron escribir



(e) Después del enganchado condicional de estrellas: $P_{8,4}$ logró escribir



(f) Después del enganchado incondicional de estrellas: $P_{6,8}$ logró escribir



(g) Después de aplicar atajos

Figura 14.14 Ilustración del algoritmo de componentes conectados: una vez terminada la parte (g), no se efectuarán enganchados en la siguiente iteración. Después de aplicar atajos, ambos componentes serán estrellas. En la última iteración no habrá cambios, y el algoritmo terminará.

Lema 14.8 El paso de enganchado incondicional de estrellas nunca engancha una estrella a otra estrella.

Demostración Supóngase que lo hace. Entonces, al principio del paso de enganchado incondicional de estrellas había dos estrellas, S_1 y S_2, que contenían los vértices i y j, respectivamente, tales que ij es una arista de G. Puesto que el enganchado condicional de estrellas no crea estrellas (lema 14.7), S_1 y S_2 eran estrellas al principio del paso de enganchado condicional de estrellas. O bien `padre[i]` $>$ `padre[j]`, en cuyo caso el árbol de i se habría enganchado (a algo) en el paso de enganchado condicional de estrellas, o `padre[j]` $>$ `padre[i]`, en cuyo caso el árbol de j se habría enganchado. Por tanto, uno de los dos, i o j, ha dejado de estar en una estrella. □

Demostración del teorema 14.4 El ciclo inicia con árboles (lema 14.6); tenemos que demostrar que ningún paso del ciclo introduce un ciclo. Es evidente que la aplicación de atajos no puede introducir un ciclo. En los pasos de enganchado, si una estrella se engancha a algún vértice peteneciente a algo que no es una estrella, no se introduce un ciclo porque las cosas que no son estrellas no están enganchadas a ninguna otra cosa. Puesto que el enganchado incondicional de estrellas siempre engancha una estrella a un vértice de algo que no es una estrella (lema 14.8), no podrá introducir un ciclo.

El enganchado condicional de estrellas sólo une la raíz de una estrella a un vértice de número más pequeño. Supóngase que v se une a w en este paso. Si w no es la raíz de su árbol, ello implica que w no está en una estrella, y en este paso no se modificará el padre de *ningún* vértice del árbol de w. Por tanto, si se forma un ciclo en el enganchado condicional de estrellas, deberá consistir en su totalidad en raíces de estrellas. Sin embargo, el enganchado condicional de estrellas sólo une una raíz a un vértice con número menor, así que es imposible que se forme tal ciclo. □

Lema 14.9 Cualquier estrella que exista al final del paso de enganchado incondicional de estrellas deberá ser un componente conectado entero.

Demostración Por el lema 14.8, la estrella ya era una estrella al principio del paso de enganchado incondicional de estrellas. Pero si cualquier vértice de la estrella estuviera adyacente (en G) a un vértice de cualquier otro árbol, el paso de enganchado incondicional de estrellas habría enganchado la estrella a otro árbol, y habría dejado de ser una estrella. □

Demostración del teorema 14.5 Puesto que los vértices de G inicialmente están en árboles disjuntos, y dos árboles sólo se enganchan si contienen vértices i y j que están adyacentes, todos los vértices de cualquier árbol dado en cualquier momento dado están en el mismo componente conectado. El algoritmo para cuando la aplicación de atajos no produce más cambios. Esto sólo puede suceder si ya no hay vértices que están a una distancia de 2 de su raíz; es decir, todos los vértices están en estrellas al final del paso de enganchado incondicional de estrellas. Por el lema 14.9, cada una de esas estrellas es un componente entero. □

14.6.3 Implementación del algoritmo en una PRAM

Algunos procesadores tienen dos "nombres". Cuando ejecutemos una operación con cada vértice (digamos, aplicación de atajos), usaremos $P_1, \ldots, P_n$ y nos referiremos a ellos como P_v. Puesto que las aristas se procesan en cada "dirección", es conveniente, por lo pronto, suponer que hay al menos $2m$ procesadores (aunque sólo se necesitarán m). Cuando ejecutemos una operación con cada arista, usaremos los primeros $2m$ procesadores y nos referiremos a ellos con los nombres P_{ij}. Puesto que las operaciones con vértices y las operaciones con aristas se efectúan en instrucciones distintas, cada procesador hace una sola cosa a la vez.

El algoritmo de PRAM supone que las entradas tienen la forma de un arreglo de aristas del grafo *G*. Cada arista aparece como dos elementos consecutivos del arreglo: la *e*-ésima arista está en `M[2*e]` y `M[2*e+1]`. El procesador P_{2e}, al tener un `pid` par, lee `M[2*e]` y luego `M[2*e+1]`. El procesador P_{2e+1}, al tener un `pid` impar, lee `M[2*e+1]` y luego `M[2*e]`. Si un procesador lee *i* y luego *j*, a partir de ese momento será el procesador para la arista (orientada) (*i*, *j*). En el programa nos referiremos a estos procesadores como P_{ij}. Así pues, cada arista tiene dedicados dos procesadores, uno para cada orientación.

La forma de las entradas no es crucial para la rapidez del algoritmo. Si las entradas se proporcionaran como matriz de adyacencia, haríamos que n^2 procesadores leyeran los elementos de la matriz en el primer paso. Los que leyeran un cero ya no efectuarían más trabajo con aristas. Hay otras variaciones aceptables del formato de las entradas.

Presentaremos otra vez el algoritmo con más pormenores de la implementación. La observación que es importante hacer aquí es que cada paso del algoritmo se puede implementar en un número constante de pasos de PRAM.

Algoritmo 14.8 Componentes conectados en paralelo

Entradas: Un arreglo de aristas del grafo, cada una de las cuales se introduce en dos posiciones consecutivas; *n*, el número de vértices; y *m*, el número de aristas.

Salidas: Un bosque de árboles dirigidos con altura máxima 1, representados por el arreglo `padre`, cuyos índices corresponden a los vértices. Cada árbol contiene los vértices de un componente conectado.

Comentarios:

1. Se usa un arreglo booleano `estrella` para registrar si un vértice está o no en una estrella; `estrella[v]` es `true` si y sólo si *v* está en una estrella. La subrutina `calcularEstrella` se da en el algoritmo 14.9.
2. La subrutina `inicCCParalelo` se da en el algoritmo 14.7.
3. La operación `enganchar` se definió en la definición 14.4.
4. La variable booleana compartida *cambio* indica si el paso de aplicación de atajos produjo algún cambio, en cada iteración del ciclo.

Procedimiento: Véase la figura 14.15. ■

Obsérvese que, en cada iteración del ciclo, un procesador P_{ij} efectúa una prueba para determinar si debe enganchar o no. A veces podría tratar de enganchar sin lograrlo, porque algún otro procesador logra escribir en `padre[padre[i]]`. En realidad, el procesador P_{ij} logra enganchar cuando más una vez durante toda la ejecución del algoritmo. Esta observación sugiere que podría ser posible acelerar el algoritmo organizando el trabajo de los procesadores de una manera más eficiente. En todo caso, como veremos, este algoritmo se ejecuta en tiempo $O(\log n)$.

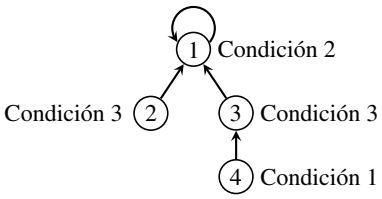
Cómo determinar si un vértice está en una estrella

Un vértice no está en una estrella si y sólo si se cumple una de las condiciones siguientes:

1. Su padre no es su abuelo,
2. Es el abuelo, pero no el padre, de algún otro vértice,
3. Su padre tiene un nieto no trivial.

```
CompsConecParalelo(G, n, m)
    // Inicialización
    Cada procesador lee n y m.
    Cada procesador calcula un número de vértice a partir de su pid:
        Para P_k tal que 1 ≤ k ≤ n, v = k:
        Para P_k fuera del intervalo anterior, v = 0.
    Cada procesador lee una arista orientada distinta:
        Para P_k, 0 ≤ k < 2m:
            Si k es par, leer M[k] y colocarlo en i, y leer M[k+1] y colocarlo en j.
            Si k es impar, leer M[k] y colocarlo en i, y leer M[k−1] y colocarlo en j.
        (Obsérvese que se asignan procesadores distintos a (i, j) y a (j, i).)


    inicCCParalelo(G, n, m);


    cambio = true;
    while (cambio == true)
       // Enganchado condicional de estrellas
       P_v ejecuta calcularEstrella(v).
       P_ij ejecuta:
       Leer padre[i], padre[j] y estrella[i].
       if (estrella[i] == true && padre[i] > padre[j])
          enganchar(i, j).


       // Enganchado incondicional de estrellas
       P_v ejecuta calcularEstrella(v).
       P_ij ejecuta:
       Leer padre[i], padre[j] y estrella[i].
       if (estrella[i] == true && padre[i] ≠ padre[j])
          // j no está en la estrella de i
          enganchar(i, j).


       // Atajos
       P_v ejecuta:
       Escribir false en cambio.
       Leer padre[v] y padre[padre[v]].
       if (padre[padre[v]] ≠ padre[v])
          Escribir padre[padre[v]] en padre[v].
          Escribir true en cambio.
       Todos los procesadores leen cambio para determinar si deben parar.
```

Figura 14.15 Procedimiento para el algoritmo 14.8



(a) Cómo saber si cada vértice está o no en una estrella

Valores iniciales del arreglo `estrella`	T T T T
Si abuelo de $v \neq$ padre de v, `estrella`[v] = F (condición 1).	T T T F
Si abuelo de $v \neq$ padre de v, `estrella`[abuelo de v] = F (condición 2).	F T T F
(En general a estas alturas, si el árbol no es una estrella, `estrella`[v] sólo puede seguir siendo T para los hijos de la raíz.)	
Si `estrella`[padre de v] es F, entonces `estrella`[v] = F (condición 3).	F F F F

(b) Cálculo para el ejemplo de la parte (a)

Figura 14.16 Cálculo del arreglo `estrella`

La figura 14.16 ilustra los tres casos y el cálculo de `estrella`. El cálculo se describe en el algoritmo que sigue, el cual obviamente tarda un tiempo constante.

Algoritmo 14.9 Cálculo de `estrella`

Comentarios: Estos pasos los ejecuta P_v (para $1 \leq v \leq n$).

```
calcularEstrella(v)
    Escribir true en estrella[v].
    Leer padre[v] y padre[padre[v]].
    if (padre[v] ≠ padre[padre[v]])
       Escribir false en estrella[v].
       Escribir false en estrella[padre[padre[v]]].
    Leer estrella[padre[v]].
    if (estrella[padre[v]] == false)
       Escribir false en estrella[v].
```

14.6.4 Análisis

Cada uno de los pasos del ciclo principal del algoritmo 14.8 se puede ejecutar en tiempo constante con una PRAM de Escritura Arbitraria, así que el número de iteraciones del ciclo determina el orden del tiempo de ejecución. Ya sólo falta demostrar que el número de iteraciones está en $O(\log n)$.

Lema 14.10 Sea h la altura de un árbol que no es estrella, antes del paso de aplicación de atajos. Después de la aplicación de atajos, su altura no puede exceder $\lfloor (h + 1)/2 \rfloor$.

Demostración El número de aristas de un camino más largo desde una hoja hasta la raíz es h. Durante la aplicación de atajos, cada sucesión de dos aristas, partiendo de la hoja, se sustituye por una sola arista. Si h es par, la longitud del camino después de la aplicación de atajos será exactamente $h/2$. Si h es impar, la longitud del camino después de la aplicación de atajos será $(h + 1)/2$. □

Definición 14.5

Para cualquier componente conectado C y $t \geq 0$, sea $h_C(t)$ la sumatoria de las alturas de todos los árboles de C al final de la t-ésima iteración del ciclo **`while`**. ■

Lema 14.11 Para cualquier componente conectado cuyos vértices no forman una estrella al principio de la t-ésima iteración del ciclo (para $t \geq 1$), $h_C(t) \leq (2/3)h_C(t - 1)$.

Demostración Consideremos qué sucede con los árboles de C durante la t-ésima iteración. Puesto que un árbol nunca se engancha a una hoja en el ciclo **`while`**, la altura de un árbol que es resultado de un enganchado no puede exceder la suma de las alturas de los dos árboles que se engancharon. Después de la aplicación de atajos, ningún árbol tiene una altura mayor que dos tercios de la altura que tenía antes, así que la suma de las alturas no puede ser mayor que dos tercios del valor que tenía antes. □

Teorema 14.12 El algoritmo 14.8 se ejecuta en tiempo $O(\log n)$ en el peor caso en una PRAM de Escritura Arbitraria con máx(n, m) procesadores.

Demostración Por el lema 14.11, para cualquier componente conectado C, tenemos

$$h_C(t - 1) \geq \frac{3}{2} h_C(t).$$

Si iteramos esta recurrencia obtenemos

$$h_C(0) \geq \left(\frac{3}{2}\right)^t h_C(t).$$

Puesto que hay n vértices en G, $h_C(t) < n$ para todos C y t, así que $h_C(0) < n$. Sea T el número de la primera iteración después de la cual los vértices de C están en una estrella. Entonces $h_C(T) = 1$. Por tanto,

$$n > h_C(0) \geq \left(\frac{3}{2}\right)^T h_C(T) = \left(\frac{3}{2}\right)^T,$$

es decir,

$$n > \left(\frac{3}{2}\right)^T.$$

Así pues,

$$T < \lg(n)/\lg(3/2).$$

Dado que T es un entero, concluimos que después de $\lfloor \lg(n)/\lg(3/2) \rfloor$ iteraciones cada uno de los componentes es una estrella. El algoritmo efectúa una sola iteración adicional, durante la cual nada cambia, así que el número total de iteraciones, y el tiempo de ejecución del algoritmo, está en $O(\log n)$.

Con sólo m procesadores, encargamos a cada uno dos aristas (orientadas). Cada paso de enganchado, incluida la inicialización, es ejecutado dos veces (en serie) por cada procesador, una vez por cada una de sus aristas. Es evidente que esto no altera el hecho de que cada paso tarda tiempo constante. □

14.7 Una cota inferior para la suma de *n* enteros

En esta sección presentaremos un argumento de cota inferior para el cómputo en paralelo. En secciones anteriores vimos que todas las variantes de la PRAM pueden sumar n enteros, o calcular el *or* de n bits, o hallar la más grande de n claves, en tiempo $O(\log n)$. Algunos modelos pueden calcular el *or* o el máximo en tiempo constante si tienen suficientes procesadores. Sin embargo, no hemos visto un algoritmo que pueda sumar n enteros en tiempo $o(\log n)$ en ninguno de estos modelos. Deduciremos una cota inferior que demuestra que tal cosa sería imposible con la PRAM de Escritura Prioritaria, el más categórico de los modelos que hemos considerado. Sea `sumaParalela` un algoritmo de PRAM para sumar n enteros ubicados en `M[0]`, ..., `M[n−1]`, el cual deja el resultado en `M[0]`. Supondremos que cada entero puede tener hasta n bits.

Varios de nuestros argumentos de cota inferior anteriores emplearon árboles de decisión. La idea en que se basan tales argumentos es que tiene que haber suficientes ramificaciones en el árbol (suficientes decisiones) para distinguir las entradas que deben generar salidas distintas. Aquí usaremos una idea similar. Una PRAM para `sumaParalela` deberá ejecutar suficientes pasos como para distinguir entre todas las posibles salidas, todas las cuales son enteros dentro del intervalo 0 a $n(2^{n-1} - 1)$. Puesto que la salida se escribe en `M[0]`, una PRAM deberá ejecutar suficientes pasos como para poder escribir en `M[0]` cualquiera de los diferentes valores. Desde luego, con una entrada dada, una PRAM siempre escribe exactamente un valor específico en `M[0]` en cualquier paso. Aquí estamos considerando el espacio de todas las entradas; contaremos todos los valores distintos que una PRAM podría escribir con todas las posibles entradas.

En realidad, a fin de simplificar el conteo, restringiremos severamente el espacio de las entradas sin restringir demasiado el intervalo de las salidas. Sólo consideraremos entradas en las que el i-ésimo entero introducido (el cual está en `M[i]`) es 2^i o bien 0. Esto nos da 2^n posibles entradas distintas (recordemos que cada entrada es una sucesión de n enteros) y *cada entrada tiene una sumatoria distinta*. Es decir, el i-ésimo bit, contando desde la derecha, es 1 en la sumatoria si y sólo si la entrada `M[i]` contenía 2^i. Por tanto, también hay 2^n posibles salidas distintas con este espacio de entradas restringido.

De hecho, en muchos modelos de PRAM un entero de n bits no cabe en una celda de memoria, y la salida tendría que escribirse en más de una celda. Para los fines del argumento de cota inferior, todas las celdas que se necesitan para contener un entero de n bits se tratan como una sola celda.

El valor que está en una celda de memoria depende de lo que los procesadores escriben (o no escriben). Lo que un procesador escribe depende del "estado" del procesador al iniciarse un paso y en lo que lee de la memoria en ese paso. Pensemos que el estado de un procesador abarca todos los aspectos internos del procesador que afectan su acción (por ejemplo, los valores de todas las variables que están en su memoria local, y su propio índice). La demostración de la cota inferior cuenta el número de estados distintos en los que el procesador puede estar, y el número de valores que se podrían escribir en celdas de memoria después de cada paso.

Teorema 14.13 Cualquier PRAM de Escritura Prioritaria con p procesadores que calcule `sumaParalela` deberá ejecutar por lo menos $\lg(m) + 1 - \lg\lg(4p)$ pasos.

Demostración Queremos contestar estas dos preguntas:

1. ¿Cuántos valores distintos puede haber en cualquier celda de memoria dada `M[i]` después de t pasos? (El intervalo de i no está restringido a las celdas de entrada.)
2. ¿En cuántos estados distintos puede estar cualquier procesador dado P_i después de t pasos?

Definimos dos sucesiones de números:

$$\begin{aligned} r_t &= r_{t-1}s_{t-1} \qquad \text{para } t > 0, \qquad r_0 = 1, \\ s_t &= p_{r_t} + s_{t-1} \qquad \text{para } t > 0, \qquad s_0 = 2, \end{aligned} \tag{14.1}$$

donde r_t y s_t tienen los significados definidos en el lema que sigue, y p es el número de procesadores. He aquí los primeros valores de las sucesiones:

t	r_1	s_t
0	1	2
1	2	$2p+2$
2	$2(2p + 2)$	$2p(2p + 2) + 2p + 2$

Después de demostrar algunos lemas, volveremos a la demostración del teorema de la cota inferior.

Lema 14.14 El número de estados distintos en que un procesador puede estar después de t pasos (considerando todas las entradas de la clase restringida que recién describimos) no puede ser mayor que r_t. El número de valores distintos que podrían estar en una celda de memoria después de t pasos (considerando todas las entradas de la clase restringida) no puede exceder s_t.

Demostración Demostraremos el lema por inducción con t. Cuando $t = 0$ (es decir, antes de que la PRAM ejecute alguna instrucción), cada procesador sólo puede estar en un estado, su estado inicial. Cada celda de memoria `M[i]` contiene uno de dos posibles valores: 0 y 2^i. Puesto que $r_0 = 1$ y $s_0 = 2$, queda establecida la base para la inducción.

Ahora, para $t > 0$, suponemos que después de $t - 1$ pasos un procesador puede estar en uno de cuando más r_{t-1} estados, y decimos que una celda de memoria puede tener uno de cuando más s_{t-1} valores. El nuevo estado de un procesador después del paso t depende del estado anterior (el estado después del paso $t - 1$) y del valor que ese procesador lee de la memoria en el paso t. Por tanto, el número de posibles estados después del paso t no puede exceder $r_{t-1}s_{t-1}$, que es r_t. En el paso t cualquier procesador puede escribir en una celda de memoria dada, y un procesador puede escribir un valor distinto para cada uno de los estados en que puede estar. Esto da pr_t posibles valores, pero también es posible que ningún procesador escriba en la celda en este paso, así que cualquiera de los s_{t-1} valores que podrían haber estado ahí antes podría seguir en la celda después del paso t. Por tanto, el número total de valores que podrían estar en una celda de memoria al término del paso t es $pr_t + s_{t-1}$, que es s_t. □

Lema 14.15 Para $t > 1$, $s_t \leq s_{t-1}^2$.

Demostración Utilizando la ecuación (14.1),

$$s_t = pr_1 + s_{t-1} = pr_{t-1}s_{t-1} + s_{t-1} = s_{t-1}(pr_{t-1} + 1) \leq s_{t-1}(pr_{t-1} + s_{t-2}) = s_{t-1}^2. \quad \square$$

Lema 14.16 Para $t \geq 1$, $s_t \leq (4p)^{2^{t-1}}$.

Demostración Para $t = 1$, $s_1 = pr_0s_0 + s_0 = 2p + 2 \leq 4p$. Para $t > 1$,

$$s_t \leq s_{t-1}^2 \leq \left((4p)^{2^{t-2}}\right)^2 = (4p)^{2^{t-1}}. \quad \square$$

Demostración del teorema 14.13 Observamos que, si cualquier algoritmo de PRAM calcula `sumaParalela` en T pasos, $s_T \geq 2^n$, porque hay 2^n salidas distintas que podrían aparecer en `M[0]` cuando el algoritmo termine. Por tanto,

$$2^n \leq s_T \leq (4p)^{2T-1}.$$

Sacamos logaritmos,

$$n \leq 2^{T-1} \lg(4p).$$

Sacamos otra vez logaritmos,

$$\lg n \leq T - 1 + \lg \lg(4p).$$

Por tanto,

$$T \geq \lg(n) + 1 - \lg \lg(4p). \quad \square$$

Corolario 14.17 Cualquier PRAM CREW, PRAM de Escritura Común, PRAM de Escritura Arbitraria o PRAM de Escritura Prioritaria que calcule `sumaParalela` deberá ejecutar por lo menos $\Theta(\log n)$ pasos si p está acotado por cualquier polinomio en n.

Demostración Cualquier programa para cualquiera de estos otros modelos es un programa válido para el modelo de Escritura Prioritaria, así que podemos usar la cota inferior del teorema 14.13. Semejante programa ejecuta al menos $\lg(n) + 1 - \lg \lg(4p)$ pasos.

Si p está acotado por un polinomio en n, entonces $\lg \lg(4p)$ está en $\Theta(\log \log n)$, y $\lg n + 1 - \lg \lg(4p)$ está en $\Theta(\log n)$. $\square$

Ejercicios

Sección 14.3 Algunos algoritmos de PRAM sencillos

14.1 Con el algoritmo 14.1, ¿qué calcula P_1 en las primeras tres iteraciones del ciclo?

14.2 Modifique el algoritmo 14.1 de modo que produzca como salida un índice de la clave más grande en lugar de la clave más grande misma. (El algoritmo modificado no deberá tener conflictos de escritura y deberá ejecutar también $\Theta(\log n)$ pasos.)

14.3 Modifique el algoritmo 14.1 de modo que sólo ejecute un paso de lectura/escritura dentro del ciclo, y que sólo use $n/2$ procesadores.

14.4 Describa una versión para PRAM del Cierre Transitivo con Atajos, algoritmo 9.1, que no tenga conflictos de escritura y se ejecute en $\Theta(\log^2 n)$ pasos. ¿Cuántos procesadores usa? *Sugerencia:* Utilice la idea para multiplicar matrices que se dio después del problema 14.1 en la sección 14.3.2.

Sección 14.4 Manejo de conflictos de escritura

14.5 Escriba un algoritmo para PRAM CREW que calcule la suma de n enteros en tiempo $O(\log n)$.

14.6 Demuestre que el *and* booleano de n bits se puede calcular en tiempo constante con una PRAM de Escritura Común (o más categórica).

14.7 Demuestre que el producto de dos matrices booleanas de $n \times n$ se puede calcular en tiempo constante con una PRAM de Escritura Común (o más categórica). (El número de procesadores deberá estar acotado por un polinomio en n.)

14.8 Describa una versión para PRAM del Cierre Transitivo por Atajos, algoritmo 9.1, que ejecute $\Theta(\log n)$ pasos en el modelo de Escritura Común (u otro más categórico). ¿Cuántos procesadores usa? *Sugerencia:* Combine la idea para multiplicar matrices que se dio después del problema 14.1 en la sección 14.3.2 con las ideas del algoritmo 14.2 y el ejercicio 14.7.

14.9 Utilizando la cota inferior planteada en la sección 14.4 (justo después del problema 14.3) para calcular el *or* de n bits en el modelo CREW, demuestre que el cálculo del máximo de n enteros tarda por lo menos un tiempo $\Omega(\log n)$ en una PRAM CREW. *Sugerencia:* Utilice la *técnica de reducción*. Demuestre que el problema "difícil" conocido (el *or* de n bits) se puede transformar en el problema actual (el mayor de n enteros) con gran rapidez (en tiempo constante) de tal manera que la respuesta al problema actual dé inmediatamente la respuesta al problema "difícil". Ahora suponga que el problema actual se puede resolver en tiempo $o(\log n)$ en una PRAM CREW y deduzca una contradicción de la cota inferior conocida. (Usamos ampliamente la técnica de reducción en el capítulo 13 en un contexto distinto, pero se trata de una técnica muy general y no es necesario haber leído el capítulo 13 para usarla en este problema.)

14.10 Utilizando la cota inferior planteada en la sección 14.4 (justo después del problema 14.3) para calcular el *or* de n bits en el modelo CREW, demuestre que la multiplicación de matrices booleanas tarda un tiempo $\Omega(\log n)$ en una PRAM CREW. *Sugerencia:* Vea la sugerencia para el ejercicio 14.9. En este caso hay que efectuar una reducción un poco más creativa.

14.11 ¿El algoritmo 14.3 funcionaría correctamente si no especificáramos cómo escoger k cuando un procesador compara dos claves iguales? Justifique su respuesta con un argumento o un contraejemplo.

14.12 Modifique el algoritmo 14.3 de modo que produzca como salida un índice de la clave más grande en lugar de la clave más grande misma. (El algoritmo modificado sólo podrá ejecutar un número constante de pasos.)

Sección 14.5 Fusión y ordenamiento

14.13 Escriba una implementación para PRAM del Ordenamiento por Inserción de n claves que se ejecute en $O(n)$ pasos de tiempo. (Puede usar cualquier variante de la PRAM, pero especifique cuál.)

14.14 Demuestre que la posición de un elemento de X y un elemento de Y no puede ser la misma en el paso de salida del algoritmo 14.4.

14.15 Modifique el algoritmo de fusión paralela (algoritmo 14.4) para fusionar dos arreglos ordenados de n y m claves, respectivamente. La indización empleada en el algoritmo 14.4 se diseñó específicamente para poder usar `fusionParalela` en el algoritmo de ordenamiento recursivo, `mergeSortParalelo` (algoritmo 14.5). Para este ejercicio, simplifique la indización escribiendo el algoritmo de fusión para arreglos con índices $0, \ldots, n-1$ y $0, \ldots, m-1$.

¿Cuántos pasos ejecuta el algoritmo modificado?

14.16 Describa un algoritmo para ordenar n claves en $O(\log n)$ pasos con una PRAM CREW. El número de procesadores puede ser mayor que n, pero deberá estar acotado por un polinomio en n. *Sugerencia:* Comience por determinar en paralelo el rango de todos los elementos. Podría serle útil en ejercicio 14.5.

⋆ **14.17** Escriba un algoritmo para fusionar dos arreglos ordenados de n claves cada uno en tiempo constante con una PRAM CREW. El número de procesadores puede ser mayor que n, pero deberá estar acotado por un polinomio en n.

Sección 14.6 Determinación de componentes conectados

⋆ **14.18** Describa la forma de combinar el algoritmo de cierre transitivo en paralelo del ejercicio 14.4 o 14.8 con otros algoritmos paralelos del capítulo para obtener un algoritmo que determine componentes conectados utilizando n^3 procesadores con un grafo de n vértices. La salida deberá ser un arreglo `lider[v]` que contenga el vértice de índice más pequeño de todo el componente conectado. Así, `lider[v] = lider[w]` si y sólo si v y w están en el mismo componente conectado. (Observe que el arreglo `lider` también se puede interpretar como un bosque adentro de árboles con altura máxima de 1, lo mismo que el arreglo `padre` en el algoritmo 14.8.)

a. ¿Con qué rapidez se ejecuta su algoritmo en el modelo de Escritura Común?

b. ¿Con qué rapidez se ejecuta su algoritmo en el modelo CREW? ¿Es necesario usar algoritmos distintos como subrutinas en el modelo CREW? ¿Cuáles?

14.19 El algoritmo de componentes conectados (algoritmo 14.8) no nos dice cuántos componentes conectados hay en el grafo de entrada G. Escriba un algoritmo paralelo para determinar el número de componentes conectados en G. Su algoritmo deberá ejecutarse en tiempo $O(\log n)$.

14.20 Utilizando los árboles de la figura 14.12(c), muestre el resultado de `enganchar`(7, 11) y (aparte) el resultado de `enganchar`(8, 11).

14.21 En el ejemplo de la figura 14.14, cuando más de un procesador trataba de escribir en una celda de memoria dada al mismo tiempo, decidimos arbitrariamente cuál lo lograba. Repita el

ejemplo tomando una decisión válida distinta en cada paso en el que haya un conflicto de escritura.

14.22 Demuestre el lema 14.6.

14.23 Suponga que en la demostración del lema 14.8 la raíz de S_1 es mayor que la de S_2. ¿S_1 necesariamente se habría enganchado a S_2? ¿Por qué?

14.24 Mostramos cómo determinar si un vértice está en una estrella. Sugiera un método para determinar (en tiempo constante) si un vértice es un solitario, para los pasos de inicialización.

14.25 Demuestre que, cuando hay conflictos de escritura, la decisión arbitraria de cuál procesador logra escribir puede tener un efecto extremo sobre el número de iteraciones del ciclo en el algoritmo 14.8. Específicamente, describa un grafo conectado G con n vértices (para n general) tal que sea posible que todos los vértices queden en una estrella después de los pasos de inicialización, pero también sea posible que el número de iteraciones del ciclo esté en $\Theta(\log n)$ para este grafo.

14.26 Sea G el grafo de entrada para el algoritmo 14.8. Sea S el conjunto de aristas (i, j) para las cuales P_{ij} logra efectuar una operación `enganchar(i, j)`. Dicho de otro modo, $(i, j) \in S$ si y sólo si P_{ij} es el procesador que realmente escribe en `padre[i]`, no simplemente uno de tal vez varios procesadores que lo intentan. Demuestre que S es una colección de árboles abarcantes (definición 8.4) para G.

14.27 Muestre cómo modificar el algoritmo 14.8 de modo que produzca una colección de árboles abarcantes para el grafo de entrada G. La salida podría adoptar la forma de una matriz booleana `caa` indizada con los pares (i, j), donde `caa[i][j] = true` indica que la arista ij está en la colección de árboles abarcantes. (Vea el ejercicio 14.26.)

★ **14.28** Este ejercicio investiga si un cambio pequeño en el algoritmo 14.8 produce un algoritmo que halle una colección mínima de árboles abarcantes para un grafo ponderado.

El paso de inicialización del algoritmo 14.8 se modifica de modo que, antes de que cada procesador lea una arista de la lista de entrada, las aristas se ordenen en orden no decreciente por peso. Suponemos que cada procesador lee la arista de la posición en la lista de entrada que corresponde a su propio índice. Así, para $k_1 < k_2$, el peso de la arista de P_{k_1} es menor o igual que el peso de la arista de P_{k_2}. El algoritmo modificado se ejecutará en una PRAM de Escritura Prioritaria. Como vimos en la sección 14.4, cuando en este modelo más de un procesador intenta escribir en la misma posición de la memoria compartida al mismo tiempo, gana el procesador con índice más bajo.

Supóngase que el algoritmo se modificó como se indica en el ejercicio 14.27, así que produce una colección de árboles abarcantes.

Demuestre que la colección de árboles abarcantes producida siempre es mínima, o bien dé un ejemplo en el que no lo sea. En el segundo caso, trate de efectuar cualesquier modificaciones adicionales del algoritmo que se necesiten para producir siempre una colección mínima de árboles abarcantes.

Problemas adicionales

14.29 La representación unaria con n bits de un entero k es una sucesión de k unos seguida de $n - k$ ceros. Para resolver cada uno de los problemas siguientes, el lector deberá usar n procesadores o menos.

a. Demuestre que una PRAM (CREW, de Escritura Común o un modelo más categórico) puede leer de `M[0]` un entero k entre 0 y n y convertirlo a su representación unaria en un paso. (La salida se colocará en las celdas `M[0]`, ..., `M[n-1]`.)

b. Demuestre que una PRAM de Escritura Prioritaria con n procesadores puede leer la representación unaria de un entero k de las celdas `M[0]`, ..., `M[n-1]` y escribir k en `M[0]` en un paso.

c. Demuestre que una PRAM CREW puede resolver el problema de la parte (b) en dos pasos.

★ **14.30** Suponga que tiene un arreglo ordenado de n claves en la memoria y p procesadores, donde p es pequeño en comparación con n. Escriba un algoritmo de PRAM CREW que busque una clave x en el arreglo. ¿Cuántos pasos ejecuta su algoritmo? *Sugerencia:* Utilice una generalización de la búsqueda binaria. Podría serle útil el ejercicio 14.29(c). Su algoritmo de búsqueda deberá ejecutar $\Theta(\log(n)/\log(p + 1))$ pasos en el peor caso.

14.31 Hojee los capítulos anteriores de este libro buscando los algoritmos que tengan una versión paralela natural. Escriba el algoritmo paralelo e indique cuántos procesadores y pasos de tiempo usa. (Escoja un algoritmo para el cual el tiempo de ejecución de la versión paralela es de orden más bajo que el de la versión secuencial.)

14.32 Prepare una lista de los problemas que se trataron en este capítulo y que están en la clase $\mathcal{NC}$ (la cual se define en la sección 14.2). ¿Algún algoritmo de este capítulo no está en $\mathcal{NC}$?

Notas y referencias

El modelo PRAM se presentó (en formas un poco distintas) en Fortune y Wyllie (1978) y Goldschlager (1978). La clase $\mathcal{NC}$ fue definida y bautizada por Steven Cook (1985) como abreviatura de "Nick's class" (la clase de Nick). El nombre se refiere a Nick Pippenger (1979). Pippenger estudió la misma clase de problemas, pero en términos de la complejidad de circuitos, más que del cómputo en paralelo. La clase tiene varias otras definiciones equivalentes.

La sección 14.5 se basa en Shiloach y Vishkin (1981). Su artículo presenta algoritmos de ordenamiento (y varios otros problemas) en los que el número de procesadores es menor que el número de claves. También incluye el algoritmo $O(\log \log n)$ para hallar la mayor de n claves que mencionamos en la sección 14.4.2, así como una solución para el ejercicio 14.29(c).

La estrategia general del algoritmo de componentes conectados que presentamos en la sección 14.6 se tomó de Hirschberg (1976). La versión rápida que presentamos aquí se basa en Shiloach y Vishkin (1982) y en Awerbuch y Shiloach (1983, 1987). Los trabajos de Awerbuch y Shiloach también contienen un algoritmo paralelo para hallar una colección mínima de árboles abarcantes (definición 8.4). La cota inferior de la sección 14.7 se basa en Beame (1986), donde se deducen más resultados generales de naturaleza similar.

Para quienes deseen leer más, la bibliografía incluye una muestra de otros artículos: Cook, Dwork y Reischuk (1986) sobre cotas superiores e inferiores para varios problemas que consideramos en las secciones 14.3 y 14.5; Chandra, Stockmeyer y Vishkin (1984) sobre varios problemas interesantes y las relaciones entre su complejidad paralela; Kruskal (1983) y Snir (1985) sobre búsquedas en paralelo (incluida la solución del ejercicio 14.30); Batcher (1968) sobre redes de ordenamiento; Landau y Vishkin (1986) sobre cotejo aproximado de cadenas; y Tarjan y Vishkin (1985) sobre componentes biconectados de grafos. Akl (1985) es un libro acerca del ordenamiento en paralelo (empleando diversos modelos de cómputo paralelo); Richards (1986), una bibliografía sobre ordenamiento en paralelo, contiene casi 400 citas. Quinn y Deo (1984) es una reseña de algoritmos paralelos para grafos. Jájá (1992) es un texto sobre algoritmos paralelos que emplea el modelo PRAM.

Greenlaw, Hoover y Ruzzo (1995) reseñan los límites conocidos de la computación en paralelo, en términos de la clase de problemas $\mathcal{P}$-completos (no confundir con los $\mathcal{NP}$-completos). La pregunta de si los problemas $\mathcal{P}$-completos son distintos de $\mathcal{NC}$ se planteó hace mucho pero todavía no puede contestarse; es el análogo paralelo de la pregunta de si la clase $\mathcal{NP}$-completa es distinta de $\mathcal{P}$, hablando de computación en serie (véase el capítulo 13).

Otro modelo importante de cómputo en paralelo requiere que los procesadores estén dispuestos en un plano y sólo estén conectados a sus vecinos; no existe memoria compartida. Este modelo se considera bueno para estudiar las capacidades de los chips VLSI (integración a muy grande escala), y suele considerarse más realista que el modelo PRAM. Ullman (1984) reseña resultados teóricos para este tipo de modelos. Hambrusch y Simon (1985) dan algunos resultados para el problema de los componentes conectados en este modelo. Parberry (1987) reseña varios modelos de computación en paralelo.

A

Ejemplos y técnicas en Java

A.1 Introducción

El propósito de este apéndice es ayudar al lector a escribir programas en Java para implementar y probar algunos algoritmos. A medida que lo lea, el lector verá que muchos aspectos no se explican con lujo de detalles. Es posible que no demos los motivos para tomar las decisiones que elegimos y que no mencionemos las alternativas. En los casos en que presentamos una solución "de recetario", ésta se puede usar tal cual para su aplicación, y no requiere modificación dependiendo del programa, o las modificaciones son directas, como una simple sustitución de nombres. Recomendamos a los lectores que deseen aprender *Java* consultar otras fuentes.

El lenguaje de programación Java adquirió fama porque se puede adaptar a las necesidades especiales de la programación para Internet. Sin embargo, Java se diseñó originalmente como lenguaje de programación de aplicación general, en consecuencia así es como lo trataremos. Pero aunque el lector no planee usar Java en una aplicación de Internet, el lenguaje tiene lazos estrechos con Internet porque ésta es la fuente primaria de gran parte de la información acerca de Java. He aquí algunos sitios de Internet (URL) que contienen información acerca de Java, versión 1.2:

```
http://java.sun.com/products/jdk/1.2/docs
http://java.sun.com/products/jdk/1.2/docs/api
http://java.sun.com/docs/books/jls/html
```

Hay muchos libros acerca de Java, pero pocos de ellos cubren todo el lenguaje, no digamos todos los paquetes que existen. No obstante, el manual de referencia es Gosling, Joy y Steele (1996).

Todo el código de este apéndice se tradujo automáticamente a partir de código probado y está protegido bajo los derechos de autor que protegen este libro. Sin embargo, ello no garantiza que esté libre de errores, y ni el autor ni la casa editorial ofrecen garantías expresas o implícitas de ningún tipo, ni asumen responsabilidad alguna por los errores u omisiones.

Java reincorpora muchas de las restricciones y verificaciones que se incluyeron en Pascal y se omitieron en C y C++. Pascal se diseñó como lenguaje de enseñanza; C y C++ se diseñaron como lenguajes de producción. Para un programador experto, tales restricciones podrían ser un fastidio y las verificaciones podrían ser innecesarias, pero para la mayoría de los estudiantes que están aprendiendo a programar las restricciones ayudan a evitar errores y las verificaciones ayudan a detectar otros errores. Por ejemplo, si en Java usamos un índice de arreglo que se sale de su intervalo, el sistema lo detectará; en este sentido Java es parecido a Pascal y diferente de C y de C++. En Java no es posible crear un apuntador a otro objeto, digamos un entero, ni se puede sumar a un apuntador, ni asignársele un valor arbitrario. Ninguna de estas operaciones está incluida en Pascal, pero todas se usan en C y C++. Con los compiladores actuales, un programa escrito en Java se ejecuta con mucha mayor lentitud que uno escrito en C. Si se trata de ejercicios para estudiantes, esto normalmente no importa mucho; la meta primaria suele tener un programa que se ejecute correctamente a cualquier velocidad.

Incluso si el lector necesita software "de producción", podría serle provechoso hacer la primera implementación en Java, porque logrará que funcione en mucho menos tiempo. Esto se denomina *creación rápida de prototipos*. La lógica y las estructuras de datos se verifican en el lenguaje que más ayuda proporciona. Luego se vuelven a codificar los procedimientos funcionales en un lenguaje más eficiente.

Un objetivo de este apéndice es cubrir suficiente material acerca de Java como para que los lectores puedan implementar los algoritmos del libro. Hay detalles latosos como entrada y salida

que es necesario precisar en un programa real. Muchos paquetes de Java lanzan excepciones, las cuales se deben manejar aunque al programador no le interesen las excepciones (como a nosotros).

Un segundo objetivo es introducir algunos recursos de Java que no mencionamos en el libro con el fin de no distraer la atención de las cuestiones algorítmicas. Se trata de recursos que la mayoría de los programadores querrá usar si se aplican seriamente al programar en Java. Aun en tales casos, optaremos por la sencillez y evitaremos algunos de los recursos más "interesantes".

A.2 Un programa principal en Java

Comenzaremos por mostrar un programa principal en la figura A.1. Este programa deberá estar en un archivo llamado `grafo.java` porque `main` pertenece a una clase llamada `grafo`. El archivo se puede compilar con el comando "`javac grafo.java`". El programa se ejecutaría con el comando "`java grafo` *archivoEntrada*". Los comandos para compilar y ejecutar se basan en un entorno Unix.[1] Como alternativa, se puede invocar el depurador de Java con `jdb` y luego teclear "`run grafo` *archivoEntrada*".

El programa lee un archivo y construye la representación de lista de adyacencia (véase la sección 7.2.3) del grafo que se define en el archivo. Se espera que el archivo tenga el número de vértices en la primera línea y tenga una arista por línea de ahí en adelante. Cada arista se especifica con los dos vértices, `de` y `a`, y podría ir seguida de un `peso`, que es de punto flotante. El formato de entrada es flexible en cuanto a que podría haber espacios o tabulaciones extra. Una vez construida la estructura de datos, se imprime su contenido; esto no resultaría práctico en el caso de un grafo grande.

Se trata de un procedimiento `main` típico; invoca subrutinas de las clases `CargarGrafo` y `BibEntrada`, que se muestran en este apéndice. Primero se verifica si está presente o no el parámetro *archivoEntrada* y, si no, el programa emite un mensaje de uso y termina. Si el parámetro está presente, el programa usa los recursos de `BibEntrada` (apéndice A.3) para obtener un objeto `BufferedReader`, `bufEntrada`, para el archivo de entrada. La clase `BufferedReader` es una clase estándar de Java. La clase `BibEntrada` oculta varios aspectos técnicos que son necesarios para acceder al archivo de entrada, y proporciona procedimientos para abrir cerrar y leer líneas de un archivo. Luego se lee la primera línea del archivo de entrada utilizando `getLine`, pero éste sólo devuelve un **`String`**, por lo que necesitamos otro procedimiento, `analizarN`, para extraer el entero de la cadena. Este entero es el número de vértices del grafo que está representado en el archivo de entrada. Casi todo el trabajo de verdad se efectúa invocando las subrutinas `inicAristas` y `cargarAristas`.

La salida se produce con los procedimientos estándar de Java `System.out.println` y `System.err.println` (empleado en algunas subrutinas para avisar de errores). En este apéndice se omite la mayor parte de los detalles de los procedimientos y clases estándar de Java; hay información completa en los sitios de Internet antes mencionados, o en un libro sobre Java. El procedimiento `println` simplemente imprime un **`String`** seguido de un carácter de salto de línea (`print` sólo imprimiría la cadena). Corresponde al programador armar la información en una cadena, pero esto se facilita mucho con el empleo de "+" para concatenar cadenas y por el hecho de que Java convierte automáticamente los números al tipo **`String`** cuando aparecen en una expresión que requiere ese tipo. Normalmente es necesario escribir un procedimiento `toString`

[1] Unix es una marca comercial de AT&T Bell Laboratories.

```
import java.io.*;

class grafo
   {
   public static
   void main(String[] argv)
      {
      int m, n;
      ListaInt[] verticesAdya;

      if (argv.length == 0)
         {
         System.out.println("Forma de uso: java grafo entrada.dat");
         System.exit(0);
         }
      String archEntrada = argv[0];
      BufferedReader bufEntrada = BibEntrada.fopen(archEntrada);
      System.out.println("Se abrió" + archEntrada + "para
       leerlo.");
      String linea = BibEntrada.getLine(bufEntrada);
      n = CargarGrafo.analizarN(linea);
      System.out.println("n = " + n);

      verticesAdya = CargarGrafo.inicAristas(n);
      m = CargarGrafo.cargarAristas(bufEntrada, verticesAdya);
      BibEntrada.fclose(bufEntrada);
      System.out.println("m = " + m);

      for (int i = 1; i <= n; i ++)
         System.out.println(i + "\t" + verticesAdya[i]);
      return;
      }
   }
```

Figura A.1 Programa en Java `grafo.java`: véase la clase `CargarGrafo` en las figuras A.2-A.4; véase la clase `BibEntrada` en las figuras A.5 y A.6.

para convertir en cadenas los objetos de clases definidas por el programador, porque el procedimiento por omisión de Java no tiene mucho sentido. No obstante, Java halla automáticamente el `toString` para la clase mediante los mecanismos de herencia; aunque el tipo de `verticesAdya[i]` es `ListaInt` en el último `println`, se imprime de manera comprensible porque la clase `ListaInt` tiene un método `toString`. En la sección A.6 se dan más detalles.

La clase `CargarGrafo` tiene varios procedimientos. Las subrutinas principales son `inicAristas` y `cargarAristas`, que se muestran en la figura A.2. La segunda construye listas de adyacencia en un ciclo simple, pero algunos detalles no muy triviales acerca de cómo se extraen los números de la línea de entrada se relegan a la subrutina `analizarArista`. Se define una clase organizadora `Arista` para la comunicación entre `cargarAristas` y `analizarArista`; la explicaremos en breve. En el ciclo de `cargarAristas`, la función `cons` crea una lista nueva pe-

```
import java.io.*;
import java.util.*;

public class CargarGrafo
    {
    public static
    ListaInt[] inicAristas(int n)
        {
        ListaInt[] verticesAdya = new ListaInt[n+1];
        for (int i = 1; i <= n; i ++)
            verticesAdya[i] = ListaInt.nil;
        return verticesAdya;
        }

    public static
    int cargarVertices(BufferedReader bufEntrada, ListaInt[] vertices-
    Adya)
        {
        int num;
        String linea;

        num = 0;
        linea = BibEntrada.getLine(bufEntrada);
        while (linea != null)
            {
            Arista aris = analizarArista(linea);
            verticesAdya[aris.de] = ListaInt.cons(aris.a,
            verticesAdya[aris.de]);
            num ++;
            linea = BibEntrada.getLine(bufEntrada);
            }
        return num;
        }
```

Figura A.2 La clase `CargarGrafo`, parte 1

gando `aris.a` al frente de la antigua lista de adyacencia de `aris.de`; luego la nueva lista se asigna como lista de adyacencia de `aris.de`.

Los procedimientos `analizarArista` de la figura A.3 y `analizarN` de la figura A.4 ilustran el uso de varios recursos de Java. Examinemos paso por paso el proceso de `analizarArista`: necesita extraer información de `linea`, construir un `Arista`, `nuevaA`, y devolverlo. La clase organizadora `Arista` tiene tres campos de ejemplar (véase la figura A.4). Obsérvese el uso de **`double`** en lugar de **`float`**; en el texto usamos **`float`** para hacerlo más comprensible, pero generalmente se prefiere **`double`** a menos que el espacio esté muy limitado, ya que tiene mayor precisión.

Primero construimos `sTok`, dándole la línea que queremos analizar, y obtenemos un objeto de la clase `stringTokenizer`. Así podremos aplicar los métodos de esa clase para obtener las

```
static
Arista analizarArista(String linea)
    {
    StringTokenizer sTok = new StringTokenizer(linea);
    int numPalabras = sTok.countTokens();
    if (numPalabras < 2 || numPalabras > 3)
        {
        System.err.println("Arista incorrecta: " + linea);
        System.exit(1);
        }

    Arista nuevaA = new Arista();
    nuevaE.de = Integer.parseInt(sTok.nextToken());
    nuevaE.a = Integer.parseInt(sTok.nextToken());
    if (numPalabras == 3)
        nuevaE.peso = Double.parseDouble(sTok.nextToken());
    else
        nuevaE.peso = 0.0;
    return nuevaE;
    }
```

Figura A.3 Continuación de la clase `CargarGrafo`, parte 2: obsérvese que `Arista` es una clase interna

palabras (fichas, o unidades lexicológicas, o *tokens*) de `linea`, además de otra información. Verificamos que la línea tenga el número requerido de palabras empleando el método `countTokens`. A continuación, el método `nextToken` extrae repetidamente la siguiente palabra, saltándose espacios y tabulaciones si es necesario. Sin embargo, `nextToken` devuelve una cadena, y lo que necesitamos es enteros y números de doble precisión.

Los tipos primitivos **`int`** y **`double`** no son clases, pero Java proporciona las clases **`Integer`** y **`Double`** (y varias más) para que los enteros y números de doble precisión disfruten de los recursos que tienen a su disposición los objetos. La clase **`Integer`** incluye el método estático `parseInt` que convierte una cadena en un **`int`**. Asimismo, **`Double`** incluye `parseDouble` para convertir una cadena en un **`double`**. Las cadenas también pueden convertirse en otros tipos primitivos.

La clase organizadora `Arista` se define en la figura A.4 como subclase de `Organizer`, para que pueda heredar la función `copy1level`, la cual se explica en el apéndice A.7. Seguiremos la regla de que cualquier clase *interna* se debe declarar `static` (sección 1.2.1); los motivos son demasiado técnicos para este apéndice.

Unas palabras acerca de la visibilidad

Como mencionamos en los primeros capítulos, Java ofrece a los programadores abundante control sobre la *visibilidad*: a qué elementos del programa se puede acceder desde otros. Si el objetivo del lector es tener un programa funcional para implementar o probar algoritmos, seguramente no querrá preocuparse acerca de esto más de lo necesario. Si todo el código se junta en un directorio y no se emplean declaraciones **`package`**, todo ese código estará en lo que se conoce como

```
public static class Arista extends Organizer
    {
    int de, a;
    double peso;

    public static Arista copy(Arista viejaA)
        { return (Arista) copy1level(viejaA); }
    }

public static
int analizarN(String linea)
    {
    StringTokenizer sTok = new StringTokenizer(linea);
    if (sTok.countTokens() != 1)
        {
        System.err.println("Línea 1 errónea: " + linea);
        System.exit(1);
        }

    int n = Integer.parseInt(sTok.nextToken());
    return n;
    }
}
```

Figura A.4 Continuación de la clase `CargarGrafo`, parte 3

el "paquete sin nombre", y se podrá acceder a todas las clases y miembros de clases desde cualquiera de ellas por omisión, sin tener que declararlas como públicas.

Para usar clases definidas en un paquete distinto, es necesario **`import`**arlas (salvo el paquete `java.lang`, que se considera tan fundamental que no es obligatorio importarlo). Las distintas figuras muestran que hemos importado los paquetes Java `io` y `util` en los archivos en que se usan una o más de sus clases.

En este apéndice hemos declarado las clases **`public`** cuando son de utilidad general, y hemos declarado **`public`** a los miembros diseñados para usarse desde otras clases. Declaramos **`protected`** a los miembros diseñados para usarse únicamente desde subclases. No obstante, se puede acceder a un miembro **`protected`** desde cualquier parte de su propio paquete. Podríamos haber declarado **`private`** (la tercera categoría de visibilidad) a esos miembros para evitar el acceso a ellos desde afuera de la clase, pero no lo hicimos. Estos ejemplos ilustran las declaraciones apropiadas para repartir los archivos entre diversos paquetes. Sin embargo, si alguien desea desarrollar paquetes deberá consultar los pormenores en otras fuentes. Todas las declaraciones **`public`** y **`protected`** se pueden omitir si todos los archivos están en el mismo directorio. En cambio, las declaraciones **`static`** sí son necesarias.

```
import java.io.*;
import java.util.*;

public class BibEntrada
   {
   static class ErrorEntrada extends Error
      {
      public ErrorEntrada(String s) { super(s); }
      }

   /** fopen abre archEntrada o System.in si archEntrada == "-". */
   public static
   BufferedReader fopen(String archEntrada)
      {
      BufferedReader bufEntrada;
      try
         {
         InputStream flujoEntrada;
         if (archEntrada.equals("-"))
            flujoEntrada = System.in;
         else
            flujoEntrada = new FileInputStream(archEntrada);

         InputStreamReader entra = new InputStreamReader(flujoEntra-
         da);
         bufEntrada = new BufferedReader(entra);
         }
      catch (java.io.IOException e)
         {
         throw new InputError(e.getMessage());
         }

      return bufEntrada;
      }
```

Figura A.5 La clase `BibEntrada` parte 1

A.3 Una biblioteca de entrada sencilla

La clase `BibEntrada` (véanse las figuras A.5 y A.6) es una clase técnica la cual podremos usar a menudo pero sin pensar casi en ella. Los lectores tendrán que consultar otras fuentes si necesitan explicaciones de la mayor parte de su funcionamiento interno. No obstante, su uso es sencillo. Invocamos `fopen` con el nombre del archivo de entrada, o la cadena `"-"` si vamos a leer de la entrada estándar. La función devuelve un objeto `BufferedReader` para ese archivo. Preferimos no examinar muy de cerca este objeto, así que nos limitamos a pasárselo a otra función, `getLine`, para leer una línea de datos del archivo. Ya vimos un ejemplo en la figura A.2. Se puede consul-

```
/** fclose cierra bufEntrada, que fopen devolvió antes. */
public static
void fclose(BufferedReader bufEntrada)
    {
    try
        {
        bufEntrada.close();
        }
    catch (java.io.IOException e)
        {
        throw new InputError(e.getMessage());
        }
    }

/** getLine lee y devuelve la siguiente línea de bufEntrada.
  * Devuelve null si se llegó al final del archivo (EOF); no hay
  * problema si se sigue invocando después de EOF.
  * Obsérvese que getLine devuelve un String sin CR (retorno
  * de carro), sea que la línea termine con CR o por EOF.
  * Así pues, no es idéntico a fgets() de C.
  */
public static
String getLine(BufferedReader bufEntrada)
    {
    String linea;
    try
        {
        linea = bufEntrada.readLine();
        }
    catch (java.io.IOException e)
        {
        throw new InputError(e.getMessage());
        }

    return linea;
    }
}
```

Figura A.6 La clase `BibEntrada`, parte 2

tar la explicación de `analizarArista` en el apéndice A.2 si se desea información acerca de la extracción de campos de datos de esta línea. Si no hay nada que leer, `getLine` devuelve **`null`**; en caso contrario devuelve la siguiente línea en forma de una sola cadena. Por último, `fclose` cierra el archivo.

El formato más común para datos es el de líneas, nosotros lo recomendamos porque ayuda a verificar que los datos tengan el formato correcto. Se pueden perder horas de frustración buscando un error en el programa que en realidad es la falta de un campo o un campo sobrante en los datos de entrada. Por otra parte, las entradas de texto (como el código de programa) *no* suele seguir el formato de líneas, por lo que `getLine` no sería la mejor opción para leer tales entradas.

Existe una técnica que nos interesa examinar con cierto detenimiento. Java contempla *excepciones* y *errores*. Si una clase "lanza (*throws*, en inglés) una excepción" y el programador quiere usar esa clase, tendrá que "manejar" la excepción. Por lo regular, el programador no desea tomarse la molestia. En tal caso, lo que la mayor parte de las fuentes recomienda es simplemente lanzarla al invocador. Sin embargo, ese invocador tendrá entonces que manejarla o lanzarla, y así.

Lo interesante es que *no es* obligatorio "manejar" los *errores*. Por tanto, nuestra recomendación para situaciones en las que se desea usar un paquete estándar de Java, y éste lanza una excepción por la que usted no desea preocuparse, es convertir la *excepción* en un *error* en el nivel más bajo posible, para que no estorbe en el código de nivel más alto. Esta técnica se demuestra en las figuras A.5 y A.6 empleando **`try`** y **`catch`**. Los procedimientos *pueden* manejar errores (también con **`try`** y **`catch`**); simplemente no es *obligatorio*. El sistema atrapará el error si ningún otro procedimiento lo hace; su programa no se perderá en el olvido. Por ello, estos procedimientos atrapan (*catch*, en inglés) las excepciones lanzadas por `BufferedReader` y las convierten en errores, con lo que sus clientes pueden hacer caso omiso de ellas o atraparlas, como gusten.

A.4 Documentación de clases de Java

Java incluye un formato de comentario especial para documentar clases, incluidas las condiciones previas y posteriores de sus métodos. Los comentarios que inician con "/**" son el principio de un comentario `javadoc`. Los lectores pueden estudiar las figuras de este apéndice y ver varios ejemplos. Este recurso es útil sobre todo para documentar un tipo de datos abstracto (TDA), porque una de las características del diseño de TDA es que la implementación debe estar encapsulada y *no debe* ser necesario examinarla para determinar cómo funciona el TDA.

El programa `javadoc` extrae estos comentarios, así como los prototipos de las funciones y procedimientos públicos, del archivo `java` en el que se implementa la clase de TDA e impone a esta información formato de HTML para que se pueda leer con un navegador de Web u otro lector de HTML. Si la clase tiene campos de ejemplar públicos, también se extrae información declarativa acerca de ellos. Existen convenciones para dar formato especial a los parámetros, así como otras formas de aliñar las salidas; consulte los detalles en la documentación de `javadoc`. Puesto que en Java no hay archivos de cabecera, como en C y C++, la mejor forma de obtener información acerca de una clase suele ser leer los archivos que `javadoc` produce con esa clase.

La colocación de los comentarios en relación con el material que están documentado podría parecer poco intuitiva. El comentario debe *preceder* al material documentado. Así, un comentario acerca de un procedimiento debe colocarse antes de la cabecera del procedimiento, pues de lo contrario `javadoc` no lo asociará a ese procedimiento. Podría ser conveniente repetir el nombre del procedimiento o campo de datos al principio de un comentario largo, para que quien lea el códi-

go sepa de qué trata el comentario sin tener que saltar hasta el final y ver qué hay ahí. Una vez que `javadoc` haya procesado el código ya no habrá problema; el nombre del procedimiento aparecerá primero, seguido de los comentarios.

A.5 Orden genérico y la interfaz "Comparable"

¿No sería bonito que pudiéramos escribir un procedimiento para ordenar y ordenar con él un conjunto de objetos de cualquier clase? Casi podemos lograrlo utilizando la interfaz **`Comparable`**. Podemos ver una "interfaz" como una especie de nombre de clase genérico. Diversas clases podrían "implementar" una interfaz proporcionando métodos con los nombres y rúbricas de tipo que esa interfaz requiere. Otros procedimientos, sin relación con esa clase, pueden invocar los métodos de la interfaz, obteniendo el método que cada clase implementa. Utilizaremos la importante interfaz **`Comparable`** como ejemplo. Muchas clases estándar de Java implementan la interfaz **`Comparable`**, como **`Integer`**, **`Double`**, **`Float`** y **`String`**. Primero mostraremos un ejemplo de clase que *usa* la interfaz **`Comparable`** como si fuera una clase, para incluir una capacidad genérica. Luego mostraremos otra clase que *implementa* la clase **`Comparable`**. Incluiremos un programa de prueba.

Las figuras A.7 y A.8 muestran una clase llamada `Ordenar`. Esta clase define cinco funciones **`boolean`** para efectuar comparaciones genéricas: `less` (menor que), `lessEq` (menor o igual que), `eq` (igual que), `greater` (mayor que) y `greaterEq` (mayor o igual que). También define una función `insertar1` que usa `lessEq` para insertar en orden un elemento nuevo en una `Lista`, sin saber nada acerca de la clase del elemento. Esto sólo funciona si los elementos que ya están en la lista son de la misma clase que el elemento nuevo a insertar, y si esa clase implementa la interfaz **`Comparable`**. En este contexto vemos que la palabra **`Comparable`** se usa como si fuera un nombre de clase. El método `compareTo` (comparar con) devuelve un **`int`** y parece ser un método de una clase llamada **`Comparable`**.

No se muestra el código de la clase `Lista`, pero es similar al de la clase `ListaInt` que se da en la sección A.6. La lógica del método `insertar1` se analizó en el ejemplo 2.1.

Si queremos que los procedimientos puedan usar los recursos de la clase `Ordenar` con una clase que estamos definiendo, deberemos especificar que nuestra clase implementa la interfaz **`Comparable`**. Mostramos un ejemplo en la figura A.9, donde se define la clase `AristaPonderada`. Obsérvese que el enunciado **`class`** incluye la frase "**`implements Comparable`**" para indicar que esta clase planea participar en esa interfaz.

La clase `AristaPonderada` es similar en algunos aspectos a la clase `Arista` de la figura A.4, pero es mucho más rica, por lo que no es una clase organizadora. No hemos hecho hincapié en los constructores no por omisión, pero eso es precisamente lo que son los dos métodos llamados `AristaPonderada` (igual que el nombre de la clase). Obsérvese que no hay tipo devuelto ni enunciado **`return`**, pero se devuelve implícitamente un nuevo objeto `AristaPonderada`. El *uso* de un constructor en Java siempre va precedido del operador **`new`**, como ilustra la figura A.10. Definir dos métodos con el mismo nombre y el mismo tipo devuelto, pero diferentes rúbricas de tipos de parámetros, se conoce como *sobrecargar*. Este recurso puede ser cómodo, pero un abuso de la sobrecarga puede inutilizar la detección de errores mediante verificación de tipos: aunque el programador haya escrito algo que no quería escribir, coincidirá con *alguna* versión del método y el error de semántica no se detectará.

```
public class Ordenar
   {
   public static
   boolean less(Comparable x, Comparable y)
      { return (x.comparateTo(y) < 0); }

   public static
   boolean lessEq(Comparable x, Comparable y)
      { return (x.comparateTo(y) <= 0); }

   public static
   boolean eq(Comparable x, Comparable y)
      { return (x.comparateTo(y) == 0); }

   public static
   boolean greater(Comparable x, Comparable y)
      { return (x.comparateTo(y) > 0); }

   public static
   boolean greaterEq(Comparable x, Comparable y)
      { return (x.comparateTo(y) >= 0); }
```

Figura A.7 La clase `Ordenar` ilustra el uso de la interfaz `Comparable` como si fuera una clase. Esta figura contiene la parte 1.

Lo fundamental para implementar la interfaz **`Comparable`** es incluir el método `compareTo`. La expresión `x.compareTo(y)` efectúa una comparación de tres vías entre x y y y devuelve un entero negativo si $x < y$, un entero positivo si $x > y$, y 0 si $x = y$. (Véanse los métodos `less`, `lessEq`, etc., en la figura A.7, que interpretan los valores devueltos.) La comparación se basa en el orden que el programador quiera definir para los objetos de esta clase. Ello implica algunos aspectos técnicos. Primero, necesitamos mutar el tipo del parámetro `e2`, de **`Object`** a `AristaPonderada`, para poder acceder a sus campos `AristaPonderada`. Eso es lo que hace la expresión "`((AristaPonderada)e2)`". Cabe señalar que son necesarios los paréntesis exteriores para obtener la precedencia correcta respecto al operador punto que sigue. (¿Qué tal si el objeto no es realmente una `AristaPonderada` y no tiene esos campos? Ello sería un error de tiempo de ejecución, y Java pararía la ejecución.)

Ahora queremos delegar la decisión al método `compareTo` de la clase **`Double`**, en lugar de tomarla de manera independiente. Hay que tener presente que **`Double`** es una clase, mientras que **`double`** es un tipo primitivo. Los tipos primitivos no tienen métodos, así que la clase **`Double`** proporciona métodos y otros recursos orientados a objetos para los objetos que son subrogados de va-

```
/** Devuelve nueva Lista con nuevoElemento insertado en orden */

public static
Lista insertar1 (Comparable nuevoElemento, Lista viejaLista)
    {
    Lista respuesta;

    if (viejaLista == Lista.nil)
        respuesta = Lista.cons(nuevoElemento, viejaLista);
    else
        {
        Comparable viejoPrimero = (Comparable)Lista.primero(viejaLista);

        if (lessEq(nuevoElemento, viejoPrimero))
            respuesta = Lista.cons(nuevoElemento, viejaLista);
        else
            {
            Lista viejoResto = Lista.resto(viejaLista);
            Lista nuevoResto = insertar1(nuevoElemento, viejoResto);
            respuesta = Lista.cons(viejoPrimero, nuevoResto);
            }
        }
    return respuesta;
    }
}
```

Figura A.8 La clase `Ordenar` ilustra el uso de la interfaz `Comparable` como si fuera una clase. Esta figura contiene la parte 2 y última.

lores **`double`**; es común calificar a estas clases como *de envoltura*. Así, creamos dos objetos de envoltura en la clase **`Double`**, utilizando los pesos de las aristas que nos interesa comparar. En última instancia, nuestro `compareTo` aplicado a objetos de tipo `AristaPonderada` simplemente ejecuta el `compareTo` de **`Double`** con los campos `peso`, y devuelve ese resultado.

Nuestra implementación permite empates: dos objetos son iguales en la comparación si tienen pesos iguales, aunque sus otros campos difieran. La documentación de Java recomienda, pero no exige, romper tales empates. Por sencillez, no lo hemos hecho aquí.

En la figura A.10 se muestra un programa de prueba para ejercitar las clases `Ordenar` y `AristaPonderada`. El programa invoca métodos de la clase `Ordenar` con tres clases diferentes de parámetros, lo que requiere tres métodos de comparación distintos, pero todos caen dentro del ámbito de **`Comparable`**, así que basta con, por ejemplo, un método `greater`. Sin la interfaz, la clase `Ordenar` necesitaría tres métodos `greater` con rúbricas de tipo distintas y cada una requeriría su propio código. Además, el apoyo estaría limitado a esos tres tipos; si se quisiera manejar cualquier clase nueva, se tendría que añadir otro procedimiento con una nueva rúbrica de tipo.

```
public class AristaPonderada implements Comparable
   {
   public int de, a;
   public double peso;

   public
   AristaPonderada(int dd, int aa, double pp)
      { de = dd; a = aa; peso = pp; }
   public
   AristaPonderada(int dd, int aa)
      { de = dd; a = aa; peso = 0.0; }
   public
   int compareTo(Object e2)
      {
      Double e1peso = new Double(peso);
      Double e2peso = newDouble( ((AristaPonderada)e2).peso);
      return e1peso.compareTo(e2peso);
      }
   public
   String toString()
      { return "(" + de + ", " + a + ", " + peso + ")"; }
   }
```

Figura A.9 La clase `AristaPonderada` implementa la interfaz **`Comparable`** y proporciona el método `compareTo` que esa interfaz requiere. También tiene dos constructores no por omisión y un método `toString`.

```
public class probOrde
    {
    public static void main(String argv[])
        {
        Integer i88 = new Integer(88), i66 = new Integer(66);
        AristaPonderada e88 = new AristaPonderada(1, 2, 88.0);
        AristaPonderada e66 = new AristaPonderada(2, 3, 66.0);
        AristaPonderada e54 = new AristaPonderada(1, 4, -54.0);
        AristaPonderada e33 = new AristaPonderada(4, 2, 33.0);
        Lista x1;

        x1 = Lista.cons(e88, Lista.nil);
        x1 = Lista.cons(e66, x1);
        x1 = Lista.cons(e54, x1);
        System.out.println(x1);
        System.out.println(e33);
        Lista x2 = Ordenar.insertar1(e33, x1);
        System.out.println(x2);
        System.out.println(Ordenar.greater("abc", "ab"));
        System.out.println(Ordenar.greater(i66, i88));
        }
    }
```

Figura A.10 El archivo `probOrde.java` prueba las funciones de la clase `Ordenar` que usan la interfaz **`Comparable`**.

A.6 Las subclases extienden la capacidad de su superclase

En esta sección mostraremos algunos recursos de Java que tienen que ver con subclases y herencia. Las explicaciones son breves y se analizan temas complejos. Mostraremos un método dinámico (no estático), `toString`, que es útil para imprimir listas, y definiremos una clase extendida, o subclase, de `ListaInt` que tiene una nueva operación con listas. Incluiremos un programa de prueba corto.

Un aspecto de las subclases que al principio parece ilógico es que una subclase tiene *más* capacidades que su superclase, que es la clase de la cual se derivó, o la clase que extiende. Para ver la lógica de esto, pensemos en las clases *persona*, *atleta* y *atleta estelar*. Los atletas son una subclase de personas, porque algunas personas no son atletas, pero todos los atletas sí son personas. Por otra parte, los atletas tienen capacidades que no toda persona tiene. Asimismo, los atletas estelares son una subclase de los atletas, pero tienen más capacidades. Esto no quiere decir que alguna persona individual no tenga capacidades de las que los atletas estelares carecen; lo único que implica es que cualquier capacidad presente en *todas* las personas está presente en todos los atletas estelares.

La figura A.11 repite las definiciones de clases de `ListaInt` que se dieron en la sección 2.3.2, pero sin los comentarios. Hemos hecho un cambio: los campos de ejemplar se declaran **`protected`**, para que las subclases puedan acceder a ellos. A continuación se define el método

`toString`, que convierte un objeto `ListaInt` en un **`String`**. Java convierte cualquier objeto en un **`String`** si aparece en un contexto que requiere ese tipo. Java proporciona un método por omisión llamado `toString` que todas las clases heredan. Una clase podría *suplantar* el método por omisión definiendo su propio método `toString`, y eso es lo que hacemos en la figura A.11. Un método de una subclase *suplanta* un método de una superclase si tiene el mismo nombre, el mismo tipo devuelto y la misma rúbrica de tipos de parámetros.

Cabe señalar que `toString` es una "envoltura" de la función recursiva `toStringR`. Java convierte automáticamente los elementos de la lista (devueltos por `primero`) en cadenas utilizando el método `toString` de la clase de ese objeto, así que la técnica funciona con listas de otros tipos de elementos; no es específica para enteros.

Digamos ahora que nos interesa añadir una nueva operación con listas, pero no queremos modificar `ListaInt`. Una opción es *extender* `ListaInt`, a una subclase que llamaremos `ListaIntA`, y definiremos la nueva operación en `ListaIntA`. Entonces, `ListaInt` será la *superclase* de `ListaIntA`. La nueva operación es un procedimiento de manipulación, `anexar1`, así que `ListaIntA` ya no es una clase no destructiva. La intención es que `anexar1` coloque un elemento nuevo al *final* de una lista existente no vacía. En términos de la implementación, se modifica el objeto de lista cuyo campo `siguiente` es `nil`. El nuevo campo `siguiente` es una lista de un elemento, el elemento nuevo. Sin embargo, el campo `siguiente` no forma parte de la interfaz del TDA, así que la especificación lógica de la acción de `anexar1` se plantea en términos de las funciones de acceso, que *sí están* en la interfaz. La figura A.12 muestra las especificaciones en forma de comentarios `javadoc`, junto con el código necesario.

Lamentablemente, es preciso redefinir los miembros **`static`** de `ListaInt` si queremos que el tipo de su resultado sea `ListaIntA`. El tipo del resultado es el tipo de un campo o el tipo devuelto por una función. Esto es válido para `nil`, `resto` y `cons` en este caso. Observe, en los casos de `nil` y `resto`, la expresión "`(ListaIntA)`". Se trata de una *mutación de tipo*, o simplemente una *mutación* (*cast*, en inglés). En nuestros ejemplos, la mutación cambia el tipo, de la superclase `ListaInt` a la subclase `ListaIntA`. La situación en el caso de `cons` es más complicada porque el método necesita construir un ejemplar de la subclase, no meramente procesar un ejemplar existente. Por ello, `cons` invoca el constructor de clase `ListaIntA` empleando el operador **`new`**. Vemos que ese constructor es similar al de `ListaInt`, pero el constructor de `ListaIntA` no quiere inventar un tipo totalmente nuevo de objeto; quiere crear un objeto parecido al objeto de la superclase. Java tiene un método especial para hacer esto: **`super`**. En nuestro ejemplo, lo único que se necesita es usar **`super`**. Si la subclase tiene campos de ejemplar adicionales que la superclase no tiene, el constructor podría incluir enunciados adicionales para inicializar esos campos.

La figura A.13 muestra un programa que prueba la clase `ListaIntA`. Obsérvese que el programa puede imprimir objetos de la clase `ListaIntA` con sólo pasárselos a `println`. Este programa ni siquiera necesita saber que existe la clase `ListaInt`; sólo trata directamente con `ListaIntA`. Sin embargo, `ListaIntA` hereda el método `toString` de `ListaInt`, y cualquier método que procese objetos `ListaIntA` podrá usarlo.

Obsérvese que, en `main`, `length` va seguido de paréntesis en un caso pero no en el otro. La razón es que `argv` es un *arreglo* (de cadenas), así que su longitud es un campo de ejemplar, mientras que `argv[0]` es un objeto **`String`**, por lo que *su* longitud es una invocación de método.

Dejamos al lector la tarea de descifrar la "acción" de `pruebaA.java`. Como pista, diremos que tiene que ver con una cola FIFO (véase la sección 2.4.2). Supóngase que se emite el comando "`java pruebaA` *palabra*". El valor de `n` es la longitud de *palabra*. Si se omite *palabra*, `n` = 0. ¿Qué orden asintótico cree el lector que tenga este programa, en función de `n`?

```
public class ListaInt
    {
    protected int elemento;
    protected ListaInt siguiente;
    public static final ListaInt nil = null;
    public static int primero(ListaInt unaLista)
        { return unaLista.elemento; }
    public static ListaInt resto(ListaInt unaLista)
        { return unaLista.siguiente; }
    public static ListaInt cons(int nuevoElemento, ListaInt viejaLista)
        { return new ListaInt(nuevoElemento, viejaLista); }
    // el verdadero constructor, pero queremos cons como interfaz.
    protected ListaInt(int nuevoElemento, ListaInt viejaLista)
        {
        elemento = nuevoElemento;
        siguiente = viejaLista;
        }

    /** Convertir ListaInt a String, similar a prolog, estilo ML. */
    public
    String toString()
        { return "[" + toStringR("", this); }

    static
    String toStringR(String prefijo, ListaInt L)
        {
        String s;

        if (L == nil)
            s = "]";
        else
            s = prefijo + primero(L) + toStringR(", ", resto(L));
        return s;
        }
    }
```

Figura A.11 El archivo `ListaInt.java` da la definición de `toString`, así como las operaciones básicas del TDA `ListaInt`. Algunos miembros son **`protected`** para que se pueda acceder a ellos desde subclases.

Pros y contras de las subclases

En el caso de una clase pequeña como `ListaInt`, no sacamos mucho provecho de la creación de una subclase; es decir, la subclase no hereda mucho. En vista de todas las complicaciones que implica definir la subclase, es razonable preguntar si vale la pena. No obstante, en otros casos hay

```
public class ListaIntA extends ListaInt
   {
   // Redefinir todos los miembros cuyo tipo de resultado se convierte
      en ListaIntA.
   public static final ListaIntA nil = (ListaIntA) ListaInt.nil;
   public static ListaIntA resto(ListaIntA unaLista)
      { return (ListaIntA) ListaInt.resto(unaLista); }
   public static ListaInt cons(int nuevoElemento, ListaIntA viejaLista)
      { return new ListaIntA(nuevoElemento, viejaLista); }
   // el verdadero constructor, pero queremos cons como interfaz.
   protected ListaIntA(int nuevoElemento, ListaIntA viejaLista)
      { super(nuevoElemento, viejaLista); }

   /** anexar1 es nuevo en la clase extendida.
     * Condición previa: unaLista != nil.
     * Condición posterior: unaLista tiene nuevoE como elemento
       adicional
     * al final, después del hasta ahora último elemento.
     * Es decir, suponiendo que antes finL era el sufijo de unaLista
     * para la cual resto(finL) == nil. Ahora primero(resto(finL)) ==
       nuevoE
     * y resto(resto(finL)) == nil.
     */
   public static
   void anexar1(ListaIntA unaLista, int nuevoE)
      {
      if (resto(unaLista) == nil)
         {
         ListaIntA nuevoUltimo = cons(nuevoE, nil);
         unaLista.siguiente = nuevoUltimo;
         }
      else
         {
         anexar1(resto(unaLista), nuevoE);
         }
      }
   }
```

Figura A.12 El archivo `ListaIntA.java` da la definición de `ListaIntA`. La palabra clave **`extends`** dice que ésta es una subclase de `ListaInt`.

más funcionalidad que heredar. La reutilización de código ayuda a garantizar la coherencia. Vimos que `ListaIntA` hereda la funcionalidad de `ListaInt`, y por ello pudo reutilizar el método `primero`.

En el lado negativo, el uso de subclases puede causar mucha confusión. Podría haber varias versiones de un método con el mismo nombre y, si no se conoce perfectamente el lenguaje, en

```
public class pruebaA
    {
    public static void main(String argv[])
        {
        int n;
        if (argv.length > 0)
            n = argv[0].length();
        else
            n = 0;
        ListaIntA a = ListaIntA.cons(1, ListaIntA.nil);
        ListaIntA fin = a;
        for (int i = 0; i < n; i ++)
            {
            ListaIntA.anexar1(fin, i+2);
            fin = ListaIntA.resto(fin);
            }
        System.out.println(a);
        System.out.println(ListaIntA.primero(a));
        System.out.println(ListaIntA.resto(a));
        }
    }
```

Figura A.13 El archivo `pruebaA.java` contiene un procedimiento `main` que ejercita la clase `ListaIntA`.

muchos casos no queda claro cuál se aplicará. Con frecuencia es necesario aplicar mutación de tipos. La implementación y prueba de algoritmos no requiere estructuras de subclase definidas por el programador, por lo que, si ésa es la meta principal del lector, le recomendamos evitarlas.

A.7 Copiado a través de la interfaz "Cloneable"

Java proporciona una interfaz llamada **`Cloneable`** que requiere el método `clone` para copiar objetos. La clase **`Objeto`** implementa `clone` como copiado de un nivel de los miembros del objeto. Es decir, en el caso de cualesquier miembros que sean también objetos, simplemente se copiarán sus *referencias*; no se clonarán recursivamente los miembros. Hemos incorporado el código técnico feo en una clase llamada `Organizer`, como se muestra en la figura A.14. El enunciado **`catch`** simplemente suprime la excepción, la cual cabe esperar que nunca se presente. Si el lector prefiere lanzar un error, le recomendamos ver el apéndice A.3.

Las clases organizadoras definidas por el programador se pueden declarar como subclases de `Organizer`, heredar `copy1level`, y usarla para implementar sus propias funciones `copy`. El caso simple se ilustra para la clase `Arista` en la figura A.4. Si la clase organizadora contiene un campo de ejemplar de otra clase organizadora, ese campo se deberá copiar explícitamente, a fin de ajustarse a las convenciones de las clases organizadoras que presentamos en la sección 1.2.2.

```
public class Organizer implements Cloneable
   {
   protected static
   Organizer copy1level(Organizer obj)
      {
      Organizer copiaObj;
      copiaObj = null;  // necesario por el try/catch
      try { copiaObj = (Organizer)obj.clone(); }
      catch(CloneNotSupportedException e) { }
      return copiaObj;
      }
   }
```

Figura A.14 El archivo `Organizer.java`

Utilizando el ejemplo `Fecha` de esa sección (`Fecha` se declararía con "**`extends`** `Organizer`", igual que `Arista`), la función `copy` para `Fecha` se convertiría en:

```
class Fecha extends Organizer
   ⋮
   public static Fecha copy(Fecha f)
      { Fecha f2 = (Fecha)copy1level(f);
        f2.año = Año.copy(f.año);  // clase organizadora
        return f2;
      }
```

Obsérvese la necesidad de la mutación de tipo "`(Fecha)`". El tipo devuelto por `copy1level` es **`Object`**.

Bibliografía

Adleman, L. M. (1994). Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266:1021-1024.

Adleman, L. M. (1998). Computing with DNA. *Scientific American*, pp. 54-61.

Aho, A. V. y Corasick, M. J. (1975). Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333-340.

Aho, A. V., Hopcroft, J. E. y Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

Aho, A. V., Hopcroft, J. E. y Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

Akl, S. (1985). *Parallel Sorting*. Academic Press, Orlando, FL.

Angluin, D. (1976). The Four Russians' Algorithm for Boolean Matrix Multiplication Is Optimal in its Class. *SIGACT News*, 8(1):29-33.

Apostolico, A. y Giancarlo, R. (1986). The Boyer-Moore-Galil String Searching Strategies Revisited. *SIAM Journal on Computing*, 15(1):98-105.

Arlazarov, V. L., Dinic, E. A., Kronrod, M. A. y Faradzev, I. A. (1970). On Economical Construction of the Transitive Closure of a Directed Graph. *Soviet Mathematics*, *Doklady*, 11(5):1209-1210.

Awerbuch, B. y Shiloach, Y. (1983). New Connectivity and MSF Algorithms for Ultracomputer and PRAM. En *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 175-179.

Awerbuch, B. y Shiloach, Y. (1987). New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258-1263.

Batcher, K. E. (1968). Sorting Networks and Their Applications. En *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 307-314.

Bayer, R. (1972). Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1(4):290-306.

Beame, P. (1986). Limits on the Power of Concurrent-Write Parallel Machines. En *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pp. 169-176.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

Bellman, R. E. y Dreyfus, S. E. (1962). *Applied Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bentley, J. L. (1982). *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, NJ.

Bentley, J. L. (1986). *Programming Pearls.* Addison-Wesley, Reading, MA.

Bentley, J. L. (columna). Programming Pearls. *Communications of the ACM.*

Bentley, J. L., Johnson, D., Leighton, F T. y McGeoch, C. C. (1983). An Experimental Study of Bin packing. En *Proceedings of the 21st Annual Allerton Conference on Communication, Control, and Computing,* pp. 51-60.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L. y Tarjan, R. E. (1973). Time Bounds for Selection. *Journal of Computer and System Sciences,* 7:448-461.

Borodin, A. y Munro, I. (1975). *Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, Nueva York.

Boyer, R. S. y Moore, J. S. (1977). A Fast String Searching Algorithm. *Communications of the ACM,* 20(10):762-772.

Brassard, G. (1985). Crusade for a Better Notation. *SIGACT News*, 17(1):60-64.

Brigham, E. O. (1974). *The Fast Fourier Transform.* Prentice-Hall, Englewood Cliffs, NJ.

Carlsson, S. (1987). A Variant of Heapsort with Almost Optimal Number of Comparisons. *Information Processing Letters,* 24(4):247-250.

Chandra, A. K., Stockmeyer, L. J. y Vishkin, U. (1984). Constant Depth Reducibility. *SIAM Journal on Computing,* 13(2):423-439.

Cook, S. A. (1971). The Complexity of Theorem Proving Procedures. En *Proceedings of the Third Annual ACM Symposiam on Theory of Computing,* pp. 151-158.

Cook, S. A. (1983). An Overview of Computational Complexity. *Communications of the ACM,* 26(6):400-408.

Cook, S. A. (1985). A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control,* 64(1-3):2-22.

Cook, S. A., Dwork, C. y Reischuk, R. (1986). Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes. *SIAM Journal on Computing,* 15(1):87-97.

Cooley, J. W. y Tukey, J. W. (1965). An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation,* 19:297-301.

Coppersmith, D. y Winograd, S. (1987). Matrix Multiplication via Arithmetic Progressions. En *Proceedings of the 19th Annual ACM Symposium on Theory of Computing,* pp. 1-6.

Cormen, T. H., Leiserson, C. E. y Rivest, R. L. (1990). *Introduction to Algorithms.* McGraw-Hill, Nueva York.

Crochemore, M. y Rytter, W. (1994). *Text Algorithms.* Oxford University Press, Nueva York.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. y Zadeck, F. K. (1991). Efficiently Computing Static Single-Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems,* 13(4):451-490.

De Millo, R. A., Lipton, R. J. y Perlis, A. J. (1979). Social Processes and Proofs of Theorems and Programs. *Communications of the ACM,* 22(5):271-280.

Deo, N. (1974). *Graph Theory with Applications to Engincering and Computer Science.* Prentice-Hall, Englewood Cliffs, NJ.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*. 1:269-271.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Dor, D. y Zwick, U. (1995). Selecting the Median. En *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorilhms*, pp. 28-37.

Dor, D. y Zwick, U. (1996a). Finding the αn-th Largest Element. *Combinatorica*, 16(1):41-58.

Dor, D. y Zwick, U. (1996b). Median Selection Requires $(2 + \epsilon)n$ Comparisons. En *Proceedings of the 37th Annual Symposium on Foundutions of Computer Science*, pp. 125-134.

Estivill-Castro, V. y Wood, D. (1996). An Adaptive Generic Sorting Algorithm that Uses Variable Partitioning. *International Journal of ComputerMathematics*, 61(3-4):181-94.

Even, S. (1973). *Algorithmic Combinatorics*. Macmillan, Nueva York.

Even, S. (1979). *Graph Algorithms*. Computer Science Press, Inc., Rockville, MD.

Fischer, M. J. (1972). Efficiency of Equivalence Algorithms. En Miller, R. y Thatcher, J., editores, *Complexity of Computer Computations*, pp. 153-167. Plenum Press, Nueva York.

Fischer. M. J. y Meyer, A. R. (1971). Boolean Matrix Multiplication and Transitive Closure. En *Conference Record*, *IEEE 12th Annual Symposium on Switching and Automata Theory*, pp. 129-131.

Floyd, R. W. (1962). Algorithm 97: Shortest Path. *Cormmunications of the ACM*, 5(6):345.

Floyd, R. W. (1964). Algorithm 245: Treesort 3. *Communications of theACM*, 7(12):701.

Ford, Jr., L. R. y Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press, Princeton, NJ.

Fortune, S. y Wyllie, J. (1978). Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 114-118.

Fredman, M. L. (1999). On the Efficiency of Pairing Heaps and Related Data Structures. *Journal of the ACM*, 46.

Fredman, M. L. y Saks, M. E. (1989). The Cell Probe Complexity of Dynamic Data Structures. En *Procredings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pp. 345-354.

Fredman, M. L., Sedgewick, R., Sleator, D. D. y Tarjan, R. E. (1986). The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica*, 1(1): l l l-129.

Fredman, M. L. y Tarjan, R. E. (1987). Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596-615.

Gabow, H. N. (1977). Two Algorithms for Generating Weighted Spanning Trees in Order. *SIAM Journal on Computing*, 6(1):139-150.

Galil, Z. (1976). Real-Time Algorithms for String-Matching and Palindrome Recognition. En *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pp. 161-173.

Galil, Z. (1979). On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm. *Communications of the ACM*, 22(9):505-508.

Galler, B. A. y Fischer, M. J. (1964). An Improved Equivalence Algorithm. *Communications of the ACM*, 7(5):301-303.

Gardner, M. (1983). *Wheels*, *Life*, *and Other Mathematical Amusements*. W. H. Freeman, San Francisco.

Garey, M. R., Graham, R. L. y Ullman, J. D. (1972). Worst-Case Analysis of Memory Allocation Algorithms. En *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pp. 143-150.

Garey, M. R. y Johnson, D. S. (1976). The Complexity of Near-Optimal Graph Coloring. *Journal of the ACM*, 23(1):43 49.

Garey, M. R. y Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*. W. H. Freeman, San Francisco.

Gehani, N. (1988). *C*: *An Advanced Introduction*, *ANSI C Edition*. Computer Science Press, Rockville, MD.

Gibbons, A. (1985). *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, England.

Goldschlager, L. M. (1978). A Unified Approach to Models of Synchronous Parallel Machines. En *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 89-94.

Gonnet, G. H. y Baeza-Yates, R. (1991). *Handbook of Algorithms and Data Structures*. AddisonWesley, Reading, MA, segunda edición.

Gosling, J., Joy, B. y Steele, G. (1996). *The Java Language Specification*. Addison-Wesley, Reading, MA.

Graham, R. L., Knuth, D. E. y Patashnik, O. (1994). *Concrete Mathematics*: *A Foundation for Computer Science*. Addison-Wesley, Reading, MA, segunda edición.

Grassmann, W. K. y Tremblay, J. P (1996). *Logic and Discrete Mathematics: A Computer Science Perspective*. Prentice-Hall, Upper Saddle River, NJ.

Greene, D. H. y Knuth, D. E. (1990). *Mathematics for the Analysis of Algorithms*. Birkhauser, Boston, tercera edición.

Greenlaw, R., Hoover, H. J. y Ruzzo, W. L. (1995). *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, Nueva York.

Gries, D. (1981). *The Science of Programming*. Springer-Verlag, Nueva York.

Guibas, L. J. y Odlyzko, A. M. (1977). A New Proof of the Linearity of the Boyer-Moore String Searching Algorithms. En *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, pp. 189-195.

Guibas, L. J. y Sedgewick, R. (1978). A Dichromatic Framework for Balanced Trees. En *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pp. 8-21.

Hall, P.A. V. y Dowling, G. R. (1980). Approximate String Marching. *Computing Surveys*, 12(4):381-402.

Hambrusch, S. E. y Simon, J. (1985). Solving Undirected Graph Problems on VLSI. *SIAM Journal on Computing*, 14(3):527-544.

Hantler, S. L. y King, J. C. (1976). An Introduction to Proving the Correctness of Programs. *ACM Computing Surveys*, 8(3):331-353.

Hirschberg, D. S. (1976). Parallel Algorithms for the Transitive Closure and the Connected Component Problems. En *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*, pp. 55-57.

Hoare, C. A. R. (1962). Quicksort. *Computer Journal*, 5(1):10-15.

Hochbaum, D. S., editor (1997). *Approximation Algorithms for* NP-*Hard Problems*. PWS Publishing Co., Boston, MA.

Hopcroft, J. E. y Tarjan, R. E. (1973a). Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135-157.

Hopcroft, J. E. y Tarjan, R. E. (1973b). Algorithm 447: Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372-378.

Hopcroft, J. E. y Tarjan, R. E. (1974). Efficient Planarity Testing. *Journal of the ACM*, 21(4):549-568.

Hopcroft, J. E. y Ullman, J. D. (1973). Set Merging Algorithms. *SIAM Journal on Computing*. 2(4):294-303.

Hyafil, L. (1976). Bounds for Selection. *SlAM Journal on Computing*, 5(1):109-114.

Ibarra, O. H. y Kim, C. E. (1975). Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM*, 22(4):463- 468.

JáJá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA.

Johnson, D. S. (1972). Fast Allocation Algorithms. En *Proceedings of the Thirteenth Annual Symposium on Switching and Automata Theory*, pp. 144-154.

Johnson, D. S. (1973). Approximation Algorithms for Combinatorial Problems. En *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pp. 38-49.

Johnson, D. S. (1974). Worst-Case Behavior of Graph Coloring Algorithms. En *Proceedings of the Fifth Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pp. 513-528. Utilitas Mathematica Publishing, Winnipeg, Canadá.

Johnson, D. S. y Trick, M. A., editores (1996). *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*., volumen 26 del *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society. Providence, RI.

Jones, D. W. (1986). An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300-311.

Kaplan, P. D., Cecchi, G. y Libchaber. A. (1995). Molecular Computation: Adlemen's Experiment Repeated. *Technical report*, NEC Research Institute, Princeton, NJ.

Kari, L., Pă un, G., Rozenberg, G., Salomaa, A. y Yu, S. (1998). DNA Computing, Sticker Systems, and Universality. *Acta Informatica*, 35(5):401-420.

Karp, R. M. (1972). Reducibility Among Combinatorial Problems. En Miller, R. y Thatcher, J., editores, *Complexity of Computer Computations*, pp. 85-104. Plenum Press, Nueva York.

Karp, R. M. (1986). Combinatorics, Complexity, and Randomness. *Communications of the ACM*, 29(2):98 109.

King, K. N. y Smith-Thomas, B. (1982). An Optimal Algorithm for Sink-Finding. *Information Processing Letters*, 14(3): 109-l l l.

Kingston, J. H. (1997). *Algorithms and Data Structures: Design, Correctness, Analysis*. AddisonWesley, Reading, MA.

Kleene, S. C. (1956). Representation of Events in Nerve Nets and Finite Automata. En Shannon, C. E. y McCarthy, J., editores, *Automata Studies*, pp. 3-40. Princeton University Press, Princeton, NJ.

Knuth, D. E. (1968). *The Art of Computer Programming*, volumen 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA.

Knuth, D. E. (1976). Big Omicron and Big Omega and Big Theta. *SlGACT News*, 8(2):18-24.

Knuth, D. E. (1984). The Complexity of Songs. *Communications of the ACM*, 27(4):344-346.

Knuth, D. E. (1997). *The Art of Computer Programming*, volumen 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA, tercera edición.

Knuth, D. E. (1998). *The Art of Computer Programming*, volumen 3: *Sorting and Searching*. Addison-Wesley, Reading, MA, segunda edición.

Knuth, D. E., Morris, Jr., J. H. y Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323-350.

Kronsjo, L. (1985). *Computational Complexity of Sequential and ParaIlel Algorithms*. Wiley, Nueva York.

Kruse, R. L., Tondo, C. L. y Leung, B. R. (1997). *Data Structures and Program Design in C*. Prentice-Hall, Upper Saddle River, NJ, segunda edición.

Kruskal, C. R. (1983). Searching, Merging, and Sorting in Parallel Computation. *IEEE Transactions on Computers*. C-32(10):942-946.

Kruskal, Jr., J. B. (1956). On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the AMS*, 7(1):48-50.

Landau, G. M. y Vishkin, U. (1986). Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm. En *Proceedings of the l 8th Annual ACM Symposium on Theory of Computing*, pp. 220-230.

Lawler, E. L., Lenstra, J. K., Kan., A. H. G. R. y Schmoys, D. B., editores (1985). *The Traveling Salesman Problem*. Wiley, Nueva York.

Linial, M. y Linial, N. (1995). Letters to *Science*. *Science*, 268:481.

Lueker, G. S. (1980). Some Techniques for Solving Recurrences. *Computing Surveys*, 12(4):419-436.

Maley, C. C. (1998). DNA Computation: Theory, Practice, and Prospects. *Evolutionary Computation*, 6(3):201-229.

Munro, I. (1971). Efficient Determination of the Transitive Closure of a Directed Graph. *Inforrnation Processing Letters*, 1(2):56-58.

Oldehoeft, R. R, Cann, D. C. y Allan, S. J. (1986). SISAL: Initial MIMD Performance Results. En Handler, W, Haupt, D. *et al*., editores, *Conference on Algorithms and, Hardware for Parallel Processing*, pp. 120-127. Springer-Verlag, Nueva York.

Pan, V. (1966). On Means of Calculating Values of Polynomials. *Russian Mathematical Surveys*, 21(1):105-136.

Parberry, I. (1987). *Parallel Complexity Theory*. Wiley, Nueva York.

Parnas, D. L. (1972). A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330-36.

Păun, G., Rozenberg, G. y Salomaa, A. (1998). *DNA Computing: New Computing Paradigms*. Springer-Verlag, Nueva York.

Perlis, A. J. (1978). The American Side of the Development of ALGOL. *SIGPLAN Notices*, 13(8):3-14.

Pippenger, N. (1979). On Simultaneous Resource Bounds. En *Proceedings of the 20th Annual Symposium of Foundations of Computer Science*, pp. 307-311. IEEE, Nueva York.

Press, W., Flannery, B., Teukolsky, S. y Vettering, W. (1988). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press.

Prim, R. C. (1957). Shortest Connection Networks and Some Generalizations. *Bell System Technical Journal*, 36:1389-1401.

Purdom, Jr., P. W. y Brown, C. A. (1985). *The Analysis of Algorithms*. Holt, Rinehart and Winston, Nueva York.

Quinn, M. J. (1987). *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Nueva York.

Quinn, M. J. y Deo, N. (1984). Parallel Graph Algorithms. *ACM Computing Surveys*, 16(3):319-348.

Rabin, M. O. (1977). Complexity of Computations. *Communications of the ACM*, 20(9):625-633.

Reingold, E. M. (1972). On the Optimality of Some Set Merging Algorithms. *Journal of the ACM*, 19(4):649-659.

Reingold, E. M., Nievergelt, J. y Deo. N. (1977). *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ.

Reingold, E. M. y Stocks, A.I. (1972). Simple Proofs of Lower Bounds for Polynomial Evaluation. En Miller, R. y Thatcher, J., editores, *Complexity: of Computer Computations*, pp. 21-30. Plenum Press, Nueva York.

Richards, D. (1986). Parallel Sorting—A Bibliography. *SIGACT News*, 18(1):28-48.

Roberts, E. (1995). *The Art and Science of C: A Library-Based Introduction to Computer Science*. Addison-Wesley, Reading, MA.

Roberts, E. (1997). *Programming Abstractions in C: A Second Course in Computer Science*. Addison-Wesley, Reading, MA.

Sahni, S. K. (1975). Approximate Algorithms for the 0/1 Knapsack Problem. *Journal of the ACM*, 22(1):115-124.

Sahni, S. K. (1976). Algorithms for Scheduling Independent Tasks. *Journal of the ACM*, 23(1):116-127.

Savage, J. E. (1974). An Algorithm for the Computation of Linear Forms. *SIAM Journal on Computing*, 3:150-158.

Schönhage, A., Paterson, M. y Pippenger, N. (1976). Finding the Median. *Journal of Computer and System Sciences*, 13:184-199.

Sedgewick, R. (1977). Quicksort with Equal Keys. *SIAM Journal on Computing*, 6(2):240-267.

Sedgewick, R. (1978). Implementing Quicksort Programs. *Communications of the ACM*, 21(10):847-857.

Sedgewick, R. (1988). *Algorithms*. Addison-Wesley, Reading, MA, segunda edición.

Sethi, R. (1996). *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, MA, segunda edición.

Sharir, M. (1981). A Strong-Connectivity Algorithm and its Application in Data Flow Analysis. *Computers and Mathematics with Applications*, 7(1):67-72.

Shell, D. L. (1959). A High-Speed Sorting Procedure. *Communications of the ACM*, 2(7):30-32.

Shiloach, Y. y Vishkin, U. (1981). Finding the Maximum, Merging and Sorting in a Parallel Computation Model. *Journal of Algorithms*, 2:88-102.

Shiloach, Y. y Vishkin, U. (1982). An $O(\log n)$ Parallel Connectivity Algorithm. *Journal of Algorithms*, 3:57-67.

Sleator, D. D. y Tarjan, R. E. (1985). Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652-686.

Smit, G. de V. (1982). A Comparison of Three String Matching Algorithms. *Software: Practice and Experience*, 12:57-66.

Snir, M. (1985). On Parallel Searching. *SIAM Journal on Computing*, 14(3):688-708.

Stasko, J. T. y Vitter, J. S. (1987). Pairing Heaps: Experiments and Analysis. *Communications of the ACM*, 30(3):234-249.

Strassen, V. (1969). Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 13:354-356.

Tarjan, R. E. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2): 146-160.

Tarjan, R. E. (1975). On the Efficiency of a Good but Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215-225.

Tarjan, R. E. (1983a). *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia.

Tarjan, R. E. (1983b). Updating a Balanced Search Tree in *O*(l) Rotations. *Information Processing Letters*, 16(5):253-257.

Tarjan, R. E. y Vishkin, U. (1985). An Efficient Parallel Biconnectivity Algorithm. *SIAM Journal on Computing*, 14(4):862-874.

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, 9(3):131-139. (solicite trabajo al autor).

Thompson, K. (1990). KQPKQ and KRPKR Endings. *ICCA Journal*, 13(4):196-199. (solicite trabajo al autor).

Thompson, K. (1991). New Results for KNPKB and KNPKN Endgames. *ICCA Journal*, 14(1):17. (solicite trabajo al autor).

Thompson, K. (1996). 6-Piece Endgames. *ICCA Journal*, 19(4):215-226. (solicite trabajo al autor).

Ullman, J. D. (1984). *Computational Aspects of VLSI*. Computer Science Press, Rockville, MD.

van Lecuwen, J. y Tarjan, R. E. (1984). Worst-Case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245-281.

Wagner, R. A. y Fischer, M. J. (1974). The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168-178.

Wainwright, R. L. (1985). A Class of Sorting Algorithms Based on Quicksort. *Communications of the ACM*, 28(4):396-403.

Warshall, S. (1962). A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11-12.

Wegner, R. (1974). Modification of Aho and Ullman's Correctness Proof of Warshall's Algorithm. *SIGACT News*, 6(1):32-35.

Weiss, M. A. (1998). *Data Structures and Problem Solving Using Java*. Addison-Wesley, Reading, MA.

Wigderson, A. (1983). Improving the Performance Guarantee for Approximate Graph Coloring. *Journal of the ACM*, 30(4):729-735.

Wilf, H. S. (1986). *Algorithms und Complexity*. Prentice-Hall, Englewood Cliffs, NJ.

Williams, J. W. J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347-348.

Winograd, S. (1970). On the Number of Multiplications Necessary to Compute Certain Functions. *Journal of Pure and Applied Mathematics*, 23:165-179.

Yao, E. (1982). Speed-Up in Dynamic Programming. *SIAM Journal on Algebraic and Discrete Methods*, 3(4):532-540.

Índice