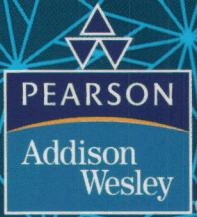
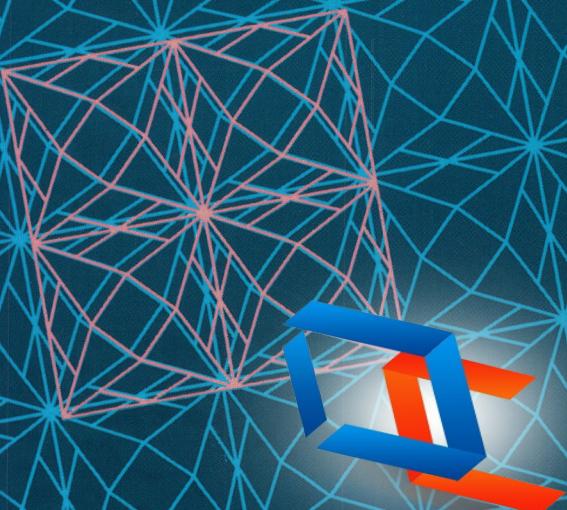


ROBERT SEDGEWICK

Algoritmos en

C++





Sitio Oficial: <https://dogramcode.com/bloglibros>

Algoritmos en C++

Sitio Oficial: <https://dogramcode.com/bloglibros>

Algoritmos en C++

Robert Sedgewick

Versión en español de

Fernando Davara Rodríguez
*Universidad Pontificia de Salamanca
Campus de Madrid, España*

Miguel Katrib Mora
Universidad de La Habana, Cuba

Sergio Ríos Aguilar
Consultor informático técnico

Con la colaboración de

Luis Joyanes Aguilar
*Universidad Pontificia de Salamanca
Campus de Madrid, España*



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Sitio Oficial: <https://dogramcode.com/bloglibros>

Versión en español de la obra titulada *Algorithms in C++*, de Robert Sedgewick, publicada originalmente en inglés por Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, © 1992.

Esta edición en español es la única autorizada.

PRIMERA EDICIÓN, 1995

© 1995 por Addison Wesley Iberoamericana, S.A.

D.R. © 2000 por ADDISON WESLEY LONGMAN DE MÉXICO, S.A. DE C.V.

Atlamulco No. 500, 5o piso
Colonia Industrial Atoto, Naucalpan de Juárez
Edo. de México, C.P. 53519

Cámara Nacional de la Industria Editorial Mexicana, Registro No. 1031.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN 968-444-401-X

Impreso en México. Printed in Mexico.

1 2 3 4 5 6 7 8 9 0 03 02 01 00 99

<https://dogramcode.com/bloglibros>

<https://dogramcode.com/bloglibros>

*A Adam, Andrew, Brett, Robbie
y especialmente a Linda*

Índice general

Prólogo	XV
----------------------	----

Fundamentos

1. Introducción	3
Algoritmos. Resumen de temas.	
2. C++ (y C)	9
Ejemplo: Algoritmo de Euclides. Tipos de datos. Entrada/Salida. Comentarios finales.	
3. Estructuras de datos elementales	17
Arrays. Listas enlazadas. Asignación de memoria. Pilas. Implementación de pilas por listas enlazadas. Colas. Tipos de datos abstractos y concretos.	
4. Árboles	39
Glosario. Propiedades. Representación de árboles binarios. Representación de bosques. Recorrido de los árboles.	
5. Recursión	55
Recurrencias. Divide y vencerás. Recorrido recursivo de un árbol. Eliminación de la recursión. Perspectiva.	
6. Análisis de algoritmos	73
Marco de referencia. Clasificación de los algoritmos. Complejidad del cálculo. Análisis del caso medio. Resultados aproximados y asintóticos. Recurrencias básicas. Perspectiva.	
7. Implementación de algoritmos	89
Selección de un algoritmo. Análisis empírico. Optimización de un programa. Algoritmos y sistemas.	

Algoritmos de ordenación

8. Métodos de ordenación elementales	103
Reglas del juego. Ordenación por selección. Ordenación por inserción. Digresión: Ordenación de burbuja. Características del rendimiento de las ordenaciones elementales. Ordenación de archivos con registros grandes. Ordenación de Shell. Cuenta de distribuciones.	

9. Quicksort	127
El algoritmo básico. Características de rendimiento del Quicksort. Eliminación de la recursión. Subarchivos pequeños. Partición por la mediana de tres. Selección.	
10. Ordenación por residuos	145
Bits. Ordenación por intercambio de residuos. Ordenación directa por residuos. Características de rendimiento de la ordenación por residuos. Una ordenación lineal.	
11. Colas de prioridad	159
Implementaciones elementales. Estructura de datos montículo. Algoritmos sobre montículos. Ordenación por montículos. Montículos indirectos. Implementaciones avanzadas.	
12. Ordenación por fusión	179
Fusión. Ordenación por fusión. Ordenación por fusión de listas. Ordenación por fusión ascendente. Características de rendimiento. Implementaciones optimizadas. Revisión de la recursión.	
13. Ordenación externa	195
Ordenación-fusión. Fusión múltiple balanceada. Selección por sustitución. Consideraciones prácticas. Fusión polifásica. Un método más fácil.	

Algoritmos de búsqueda

14. Métodos de búsqueda elementales	213
Búsqueda secuencial. Búsqueda binaria. Búsqueda por árbol binario. Eliminación. Arboles binarios de búsqueda indirecta.	
15. Árboles equilibrados	237
Árboles descendentes 2-3-4. Árboles rojinegros. Otros algoritmos.	
16. Dispersión	255
Funciones de dispersión. Encadenamiento separado. Exploración lineal. Doble dispersión. Perspectiva.	
17. Búsqueda por residuos	271
Árboles de búsqueda digital. Árboles de búsqueda por residuos. Búsqueda por residuos múltiple. Patricia.	
18. Búsqueda externa	287
Acceso secuencial indexado. Árboles B. Dispersion extensible. Memoria virtual.	

Procesamiento de cadenas

19. Búsqueda de cadenas	307
Una breve historia. Algoritmo de fuerza bruta. Algoritmo de Knuth - Morris - Pratt. Algoritmo de Boyer - Moore. Algoritmo de Rabin - Karp. Búsquedas múltiples.	
20. Reconocimiento de patrones	325
Descripción de patrones. Máquinas de reconocimiento de patrones. Representación de la máquina. Simulación de la máquina.	
21. Análisis sintáctico	337
Gramáticas libres de contexto. Análisis descendente. Análisis ascendente. Compiladores. Compilador de compiladores.	
22. Compresión de archivos	351
Codificación por longitud de series. Codificación de longitud variable. Construcción del código de Huffman. Implementación.	
23. Criptología	365
Reglas del juego. Métodos elementales. Máquinas de cifrar/descifrar. Sistemas de cripto de claves públicas.	

Algoritmos geométricos

24. Métodos geométricos elementales	379
Puntos, líneas y polígonos. Intersección de segmentos de líneas. Camino cerrado simple. Inclusión en un polígono. Perspectiva.	
25. Obtención del cerco convexo	391
Reglas del juego. Envolventes. La exploración de Graham. Eliminación interior. Rendimiento.	
26. Búsqueda por rango	407
Métodos elementales. Método de la rejilla. Árboles bidimensionales. Búsqueda por rango multidimensional.	
27. Intersección geométrica	423
Segmentos horizontales y verticales. Implementación. Intersección de segmentos en general.	
28. Problemas del punto más cercano	435
Problema del par más cercano. Diagramas de Voronoi.	

Algoritmos sobre grafos

29.	Algoritmos sobre grafos elementales	451
	Glosario. Representación. Búsqueda en profundidad. Búsqueda en profundidad no recursiva. Búsqueda en amplitud. Laberintos. Perspectivas.	
30.	Conectividad	475
	Componentes conexas. Biconectividad. Algoritmos de unión - pertenencia.	
31.	Grafos ponderados	491
	Árbol de expansión mínimo. Búsqueda en primera prioridad. Método de Kruskal. El camino más corto. Árbol de expansión mínimo y camino más corto en grafos densos. Problemas geométricos.	
32.	Grafos dirigidos	513
	Búsqueda en profundidad. Clausura transitiva. Todos los caminos más cortos. Ordenación topológica. Componentes fuertemente conexas.	
33.	Flujo de red	529
	El problema del flujo de red. Método de Ford - Fulkerson. Búsqueda de red.	
34.	Concordancia	539
	Grafos bipartidos. Problema del matrimonio estable. Algoritmos avanzados.	

Algoritmos matemáticos

35.	Números aleatorios	555
	Aplicaciones. Método de congruencia lineal. Método de congruencia aditiva. Comprobación de la aleatoriedad. Notas de implementación.	
36.	Aritmética	569
	Aritmética polinómica. Evaluación e interpolación polinómica. Multiplicación polinómica. Operaciones aritméticas sobre enteros grandes. Aritmética de matrices.	
37.	Eliminación gaussiana	585
	Un ejemplo simple. Esbozo del método. Variaciones y extensiones.	
38.	Ajuste de curvas	597
	Interpolación polinómica. Interpolación spline. Método de los mínimos cuadrados.	

39. Integración	609
Integración simbólica. Métodos de cuadratura elementales. Métodos compuestos. Cuadratura adaptativa.	

Temas avanzados

40. Algoritmos paralelos	623
Aproximaciones generales. Mezcla perfecta. Arrays sistólicos. Perspectiva.	
41. La transformada rápida de Fourier	637
Evaluar, multiplicar, interpolar. Raíces complejas de la unidad. Evaluación de las raíces de la unidad. Interpolación en las raíces de la unidad. Implementación.	
42. Programación dinámica	649
El problema de la mochila. Producto de matrices en cadena. Árboles binarios de búsqueda óptima. Necesidades de espacio y tiempo.	
43. Programación lineal	661
Programas lineales. Interpretación geométrica. El método simplex. Implementación.	
44. Búsqueda exhaustiva	677
Búsqueda exhaustiva en grafos. Vuelta atrás. Digresión: Generación de permutaciones. Algoritmos de aproximación.	
45. Problemas NP-completos	691
Algoritmos deterministas y no deterministas de tiempo polinómico. Compleción NP. El teorema de Cook. Algunos problemas NP-completos.	
Epílogo	701
Vocabulario técnico bilingüe	705
Índice de programas	713
Índice analítico	719

Prólogo

La finalidad de este libro es dar una idea clara de los algoritmos más importantes empleados hoy día en las computadoras y enseñar sus técnicas fundamentales a quienes, cada vez en mayor número, tienen necesidad de conocerlos. Se puede utilizar como libro de texto para segundo, tercero o cuarto curso de informática, una vez que los estudiantes hayan adquirido cierta habilidad en la programación y se hayan familiarizado con los sistemas informáticos, pero antes de realizar cursos de especialización en áreas avanzadas de la informática o de sus aplicaciones.

Además, el libro puede ser útil para la autoformación o como texto de consulta para aquellos que trabajan en el desarrollo de aplicaciones o sistemas para computadoras, ya que contiene un gran número de algoritmos útiles e información detallada sobre sus características de ejecución. La amplia perspectiva adoptada en el libro lo convierte en una adecuada introducción a este campo.

Los algoritmos se expresan en el lenguaje de programación C++ (también se dispone de versiones del libro en Pascal y C), pero no se necesitan conocimientos de un lenguaje de programación específico —el tratamiento aquí contemplado es autónomo, aunque un tanto rápido—. Los lectores que estén familiarizados con C++ encontrarán en este lenguaje un vehículo útil para aprender una serie de métodos de interés práctico. Aquellos que tengan conocimientos de algoritmos básicos encontrarán en el libro una forma útil de conocer diversas características del lenguaje C++, y simultáneamente aprenderán algunos algoritmos nuevos.

Finalidad

El libro contiene 45 capítulos agrupados en ocho partes principales: fundamentos, ordenación, búsqueda, procesamiento de cadenas, algoritmos geométricos, algoritmos sobre grafos, algoritmos matemáticos y temas avanzados. Un objetivo importante a la hora de escribir este libro ha sido reunir los métodos fundamentales de diversas áreas, con la finalidad de dar a conocer los más empleados en la resolución de problemas por medio de computadoras. Algunos de los capítulos son una introducción a materias más avanzadas. Se espera que las descripciones aquí empleadas permitan al lector comprender las propiedades básicas de algoritmos fundamentales, que abarcan desde las colas de prioridad y la dispersión, al simplex y la transformada de Fourier rápida.

Se aconseja que el lector haya realizado previamente uno o dos cursos de informática, o disponga de una experiencia equivalente en programación, para que pueda valorar el contenido de este libro: lo ideal sería un curso de programación en un lenguaje de alto nivel como C++, C o Pascal, y quizás otro curso sobre conceptos fundamentales de sistemas de programación. Este libro está pues destinado a cualquier persona con experiencia en un lenguaje moderno de programación y con las ideas básicas de los sistemas modernos de computadora. Se incluyen en el texto algunas referencias que pueden ayudar a subsanar las posibles lagunas del lector.

En su mayor parte, los conceptos matemáticos que sustentan los resultados analíticos se explican (o bien se clasifican como «más allá de la finalidad» de este libro), por lo que para la comprensión general del libro no se requiere una preparación específica en matemáticas, aunque, en definitiva, es útil una cierta madurez matemática. Algunos de los últimos capítulos tratan algoritmos relacionados con conceptos matemáticos más avanzados —se han incluido para situar a los algoritmos en el contexto de otros métodos y no para enseñar los conceptos matemáticos—. Por lo tanto, la presentación de los conceptos matemáticos avanzados es breve, general y descriptiva.

Utilización en planes de estudio

La forma en que se puede enseñar esta materia es muy flexible. En gran medida, se pueden leer unos capítulos independientemente de otros, aunque en algunos casos los algoritmos de un capítulo utilizan los métodos del capítulo anterior. Se puede adaptar el texto para la realización de diferentes cursos mediante la posible selección de 25 ó 30 de los 45 capítulos, según las preferencias del profesor y la preparación de los estudiantes.

El libro comienza con una sección de introducción a las estructuras de datos y al diseño y análisis de algoritmos. Esto establece las pautas para el resto de la obra y proporciona una estructura, dentro de la que se tratan algoritmos más avanzados. Algunos lectores pueden saltarse u hojear esta sección; otros pueden aprender aquí las bases.

En un curso elemental sobre «algoritmos y estructuras de datos» podrían omitirse algunos de los algoritmos matemáticos y ciertos temas avanzados, haciendo hincapié en la forma en la que se utilizan las estructuras de datos en las implementaciones. En un curso intermedio sobre «diseño y análisis de algoritmos» podrían omitirse algunas de las secciones que están orientadas a la práctica y recalcarse la identificación y el estudio de las condiciones en las que los algoritmos alcanzan rendimientos asintóticos satisfactorios. En un curso sobre las «herramientas del software» se podrían omitir las matemáticas y el material algorítmico avanzado, y así poner mayor énfasis en cómo integrar las implementaciones propuestas en grandes sistemas o programas. Un curso sobre «al-

gorítmos» podría adoptar un enfoque de síntesis e introducir conceptos de todas estas áreas.

Algunos profesores, para dar una orientación particular, pueden añadir material complementario a los cursos descritos anteriormente. Para las «estructuras de datos y algoritmos» se podría ampliar el estudio sobre estructuras básicas de datos; para «diseño y análisis de algoritmos» se podría profundizar en el análisis matemático, y para las «herramientas del software» convendría profundizar en las técnicas de ingeniería del software. En este libro se contemplan todas estas áreas, pero el énfasis se pone en los algoritmos propiamente dichos.

En los últimos años se han empleado versiones anteriores de este libro en decenas de colegios y universidades norteamericanas, como texto para el segundo o tercer curso de informática y como lectura complementaria para otros cursos. En Princeton, la experiencia ha demostrado que el amplio espectro que cubre este libro proporciona a los estudiantes de los últimos cursos una introducción a la informática, que puede ampliarse con cursos posteriores sobre análisis de algoritmos, programación de sistemas e informática teórica, a la vez que proporciona a todos los estudiantes un gran conjunto de técnicas de las que pueden obtener un provecho inmediato.

Hay 450 ejercicios, diez por capítulo, que generalmente se dividen en dos grupos. La mayor parte tienen como finalidad comprobar que los estudiantes han entendido la materia del libro, y hacer que trabajen en un ejemplo o apliquen los conceptos descritos en el texto. Sin embargo, algunos de ellos son para implementar y agrupar algoritmos, necesitándose en ocasiones estudios empíricos para comparar algoritmos y conocer sus propiedades.

Algoritmos de uso práctico

Este libro está orientado al tratamiento de algoritmos de uso práctico. El objetivo es enseñar a los estudiantes las herramientas que tienen a su alcance, para que puedan implementar con absoluta confianza algoritmos útiles, ejecutarlos y depurarlos. Se incluyen en el texto implementaciones completas de los métodos empleados, junto con descripciones del funcionamiento de los programas en un conjunto coherente de ejemplos. De hecho, como se verá en el epílogo, se incluyen cientos de figuras que han sido generadas por los propios algoritmos. Muchos algoritmos se aclaran desde un punto de vista intuitivo a través de la dimensión visual que proporcionan las figuras.

Las características de los algoritmos y las situaciones en que podrían ser útiles se presentan de forma detallada. Aunque no se haga hincapié en ellas, no se ignoran las relaciones entre el análisis de algoritmos y la informática teórica. Cuando se considere apropiado, se presentarán los resultados analíticos y empíricos para ilustrar por qué se prefieren ciertos algoritmos. Cuando sea interesante, se describirá la relación entre los algoritmos prácticos presentados y los resultados puramente teóricos. Se encontrará a lo largo del texto información

específica sobre las características de rendimiento de los algoritmos, bajo la forma de «propiedades», que resumen los hechos importantes de los algoritmos que merecen un estudio adicional.

Algunos algoritmos se utilizan en programas relativamente pequeños para resolver problemas concretos y otros se integran, como parte de un todo, en sistemas relativamente grandes. Muchos algoritmos fundamentales encuentran aplicación en ambos casos. Se indicará cómo adaptar algoritmos específicos para resolver problemas concretos o algoritmos generales para su integración en programas más grandes. Tales consideraciones son particularmente interesantes para los algoritmos expresados en un lenguaje orientado a objetos, tal como C++. En este libro se proporciona la información apropiada que puede utilizarse para hacer intercambios inteligentes entre utilidad y rendimiento en implementaciones de algoritmos muy utilizados.

A pesar de que existe un escaso tratamiento directo del empleo específico de los algoritmos en aplicaciones para la ciencia y la ingeniería, las posibilidades de tal uso se mencionarán cuando sea conveniente. La experiencia demuestra que cuando los estudiantes aprenden pronto buenos algoritmos informáticos son capaces de aplicarlos para resolver problemas a los que se enfrentarán más adelante.

Lenguaje de programación

El lenguaje de programación utilizado a lo largo de este libro es C++ (también existen versiones en Pascal y C). Cualquier lenguaje particular tiene ventajas e inconvenientes —la intención aquí es facilitar, al creciente número de personas que utilizan el C++ como lenguaje original para sus aplicaciones, el acceso a los algoritmos fundamentales que se han ido desarrollando a través de los años—. Los programas se pueden traducir fácilmente a otros lenguajes de programación modernos, ya que están escritos en una forma sencilla que los hace relativamente independientes del lenguaje. Desde luego, muchos de los programas han sido traducidos desde Pascal, C y otros lenguajes, aunque se intenta utilizar el lenguaje C estándar cuando sea apropiado. Por otra parte, C++ se adapta perfectamente a la tarea del libro, dado su soporte básico en la abstracción de datos y su programación modular que permite expresar claramente las relaciones entre las estructuras de datos y los algoritmos.

Algunos de los programas se pueden simplificar utilizando aspectos más avanzados del lenguaje, pero esto ocurre menos veces de las que se podría pensar. Aunque las características del lenguaje se presentarán cuando sea apropiado, este libro no tiene como objetivo ser un manual de referencia de C++ o de la programación orientada a objetos. Mientras se utilizan las clases de C++ reiteradamente, no se usan plantillas, herencias, ni funciones virtuales, pero los algoritmos están codificados así para facilitar los procesos de instalación en sistemas grandes, donde tales aspectos se pueden utilizar para beneficiarse de la

programación orientada a objetos. Cuando se precise hacer una elección será concentrándose en los algoritmos, no en los detalles de la implementación ni en las características del lenguaje.

Una de las metas de este libro es presentar los algoritmos de la forma más simple y directa que sea posible. Los programas no están para leerse por sí mismos, sino como parte del texto que los encuadra. Este estilo se ha elegido como una alternativa a, por ejemplo, la introducción de comentarios entre líneas. El estilo es coherente, de forma que programas similares parecerán similares.

Agradecimientos

Mucha gente me ha ayudado al comentar las versiones anteriores de este libro. En particular, los estudiantes de Princeton y Brown han sufrido con las versiones preliminares del material del libro en los ochenta. En especial, doy las gracias a Trina Avery, Tom Freeman y Janet Incerpi por su ayuda en la producción de la primera edición. En particular a Janet por pasar el libro al formato T_EX, añadir los miles de cambios que hice después del «último borrador» de la primera edición, guiar los archivos a través de diversos sistemas para imprimir las páginas e incluso escribir una rutina de revisión para T_EX utilizada para obtener manuscritos de prueba, entre otras muchas cosas. Solamente cuando yo mismo desempeñé estas tareas en posteriores versiones, pude apreciar realmente la contribución de Janet. Me gustaría también dar las gracias a muchos de los lectores que me ayudaron con comentarios detallados sobre la segunda edición, entre ellos a Guy Almes, Jay Gischer, Kennedy Lemke, Udi Manber, Dana Richards, John Reif, M. Rosenfeld, Stephen Seidman y Michael Quinn.

Muchos de los diseños de las figuras están basados en el trabajo conjunto con Marc Brown en el proyecto «aula electrónica» en la Brown University en 1983. Agradezco el apoyo de Marc y su ayuda en la creación de los diseños (sin mencionar el sistema con el que trabajábamos). También me gustaría agradecer la ayuda de Sarantos Kapidakis para obtener el texto final.

Esta versión C++ debe su existencia a la tenacidad de Keith Wollman, quien me convenció para realizarla, y a la paciencia de Peter Gordon, que estaba convencido de que la sacaría adelante. La buena voluntad de Dave Hanson para contestar preguntas acerca de C y C++ fue incalculable. También me gustaría agradecer a Darcy Cotten y a Skip Plank su ayuda para producir el libro.

Mucho de lo que he escrito aquí lo he aprendido gracias a las enseñanzas de Don Knuth, mi consejero en Stanford. Aunque Don no ha tenido influencia directa sobre este trabajo se puede sentir su presencia en el libro, porque fue él quien supo colocar el estudio de algoritmos sobre una base científica de tal forma que sea posible realizar un trabajo como éste.

Estoy muy agradecido por el apoyo de la Brown University e INRIA donde realicé la mayor parte del trabajo del libro, y al Institute for Defense Analyses y al Xerox Palo Alto Research Center, donde hice parte del libro mientras lo vi-

sitaba. Muchas partes del libro se deben a la investigación realizada y cedida generosamente por la National Science Foundation y la Office of Naval Research. Finalmente, quisiera dar las gracias a Bill Bowen, Aaron Lemonick y Neil Rudenstine de la Princeton University por apoyarme al crear un entorno académico en el que fui capaz de preparar este libro, a pesar de tener muchas otras responsabilidades.

ROBERT SEDGEWICK

Marly-le-Roi, Francia, febrero, 1983

Princeton, New Jersey, enero, 1990

Princeton, New Jersey, enero, 1992

Fundamentos

Introducción

El objetivo de este libro es estudiar una variedad muy extendida de *algoritmos* útiles e importantes: los métodos de resolución de problemas adaptados para su realización por computadora. Se tratarán diferentes áreas de aplicación, poniendo siempre especial atención a los algoritmos «fundamentales» cuyo conocimiento es importante e interesante su estudio. Dado el gran número de algoritmos y de dominios a cubrir, muchos de los métodos no se estudiarán en profundidad. Sin embargo, se tratará de emplear en cada algoritmo el tiempo suficiente para comprender sus características esenciales y para respetar sus particularidades. En resumen, la meta es aprender un gran número de los algoritmos más importantes que se utilizan actualmente en computadoras, de forma que se pueda utilizarlos y apreciarlos.

Para entender bien un algoritmo, hay que realizarlo y ejecutarlo; por consiguiente, la estrategia recomendada para comprender los programas que se presentan en este libro es implementarlos y probarlos, experimentar con variantes y tratar de aplicarlos a problemas reales. Se utilizará el lenguaje de programación C++ para presentar y realizar la mayor parte de los algoritmos; no obstante, al utilizar sólo un subconjunto relativamente pequeño del lenguaje, los programas pueden traducirse fácilmente a otros lenguajes de programación.

Los lectores de este libro deben poseer al menos un año de experiencia en lenguajes de programación de alto y bajo nivel. También sería conveniente tener algunos conocimientos sobre los algoritmos elementales relativos a las estructuras de datos simples tales como arrays, pilas, colas y árboles, aunque estos temas se traten detalladamente en los Capítulos 3 y 4. De igual forma, se suponen unos conocimientos elementales sobre la organización de la máquina, lenguajes de programación y otros conceptos elementales de informática. (Cuando corresponda se revisarán brevemente estas materias, pero siempre dentro del contexto de resolución de problemas particulares.) Algunas de las áreas de aplicación que se abordarán requieren conocimientos de cálculo elemental. También se utilizarán algunos conceptos básicos de álgebra lineal, geo-

metría y matemática discreta, pero no es necesario el conocimiento previo de estos temas.

Algoritmos

La escritura de un programa de computadora consiste normalmente en implementar un método de resolución de un problema, que se ha diseñado previamente. Con frecuencia este método es independiente de la computadora utilizada: es igualmente válido para muchas de ellas. En cualquier caso es el método, no el programa, el que debe estudiarse para comprender cómo está siendo abordado el problema. El término *algoritmo* se utiliza en informática para describir un método de resolución de un problema que es adecuado para su implementación como programa de computadora. Los algoritmos son la «esencia» de la informática; son uno de los centros de interés de muchas, si no todas, de las áreas del campo de la informática.

Muchos algoritmos interesantes llevan implícitos complicados métodos de organización de los datos utilizados en el cálculo. Los objetos creados de esta manera se denominan *estructuras de datos*, y también constituyen un tema principal de estudio en informática. Así, estructuras de datos y algoritmos están intimamente relacionados; en este libro se mantiene el punto de vista de que las estructuras de datos existen como productos secundarios o finales de los algoritmos, por lo que es necesario estudiarlas con el fin de comprender los algoritmos. Un algoritmo simple puede dar origen a estructuras de datos complicadas, y a la inversa, un algoritmo complicado puede utilizar estructuras de datos simples. En este libro se estudian las propiedades de muchas estructuras de datos, por lo que bien se podría haber titulado *Algoritmos y estructuras de datos en C++*.

Cuando se desarrolla un programa muy grande, una gran parte del esfuerzo se destina a comprender y definir el problema a resolver, analizar su complejidad y descomponerlo en subprogramas más pequeños que puedan realizarse fácilmente. Con frecuencia sucede que muchos de los algoritmos que se van a utilizar son fáciles de implementar una vez que se ha descompuesto el programa. Sin embargo, en la mayor parte de los casos, existen unos pocos algoritmos cuya elección es crítica porque su ejecución ocupará la mayoría de los recursos del sistema. En este libro se estudiará una variedad de algoritmos fundamentales básicos para los grandes programas de muchas áreas de aplicación.

El compartir programas en los sistemas informáticos es una técnica cada vez más difundida, de modo que aunque los usuarios serios *utilizarán* íntegramente los algoritmos de este libro, quizás necesiten *implementar* sólo alguna parte de ellos. Realizando las versiones simples de los algoritmos básicos se podrá comprenderlos mejor y también utilizar versiones avanzadas de forma más eficaz. Algunos de los mecanismos de software compartido de los sistemas de computadoras dificultan a menudo la adaptación de los programas estándar a la reso-

lución eficaz de tareas específicas, de modo que muchas veces surge la necesidad de reimplementar algunos algoritmos básicos.

Los programas están frecuentemente sobreoptimizados. Puede no ser útil esmerarse excesivamente para asegurarse de que una realización sea lo más eficiente posible, a menos que se trate de un algoritmo susceptible de utilizarse en una tarea muy amplia o que se utilice muchas veces. En los otros casos, bastará una implementación relativamente simple; se puede tener cierta confianza en que funcionará y en que posiblemente su ejecución sea cinco o diez veces más lenta que la mejor versión posible, lo que significa unos pocos segundos extra en la ejecución. Por el contrario la elección del algoritmo inadecuado desde el primer momento puede producir una diferencia de un factor de unos cientos, o de unos miles, o más, lo que puede traducirse en minutos, horas, o incluso más tiempo de ejecución. En este libro se estudiarán implementaciones razonables y simples de los mejores algoritmos.

A menudo varios algoritmos diferentes son válidos para resolver el mismo problema. La elección del mejor algoritmo para una tarea particular puede ser un proceso muy complicado y con frecuencia conllevará un análisis matemático sofisticado. La rama de la informática que estudia tales cuestiones se llama *análisis de algoritmos*. Se ha demostrado a través de dicho análisis que muchos de los algoritmos que se estudiarán tienen un rendimiento muy bueno, mientras que de otros se sabe que funcionan bien simplemente a través de la experiencia. No se hará hincapié en comparaciones de resultados de rendimiento: la meta es aprender algunos algoritmos que resuelvan tareas importantes. Pero, como no se debe usar un algoritmo sin tener alguna idea de qué recursos podría consumir, se intentará precisar de qué forma se espera que funcionen los algoritmos de este libro.

Resumen de temas

A continuación se presenta una breve descripción de las partes principales del libro, que enuncian algunos de los temas específicos, así como también alguna indicación de la orientación general sobre la materia. Este conjunto de temas intentará tocar tantos algoritmos fundamentales como sea posible. Algunos de los temas tratados constituyen el «corazón» de diferentes áreas de la informática, y se estudiarán en profundidad para comprender los algoritmos básicos de gran utilidad. Otras áreas son campo de estudios superiores dentro de la informática y de sectores relacionados con ella, tales como el análisis numérico, la investigación operativa, la construcción de compiladores y la teoría de algoritmos —en estos casos el tratamiento servirá como una introducción a dichos campos a través del examen de algunos métodos básicos—.

En el contexto de este libro, los *FUNDAMENTOS* son las herramientas y métodos que se utilizarán en los capítulos posteriores. Se incluye una corta discusión de C++, seguida por una introducción a las estructuras de datos básicas,

que incluye arrays, listas enlazadas, pilas, colas y árboles. Se presentará la utilización práctica de la recursión, encaminando el enfoque básico hacia el análisis y la realización de algoritmos.

Los métodos de *ORDENACIÓN*, para reorganizar archivos en un orden determinado, son de vital importancia y se tratan en profundidad. Se desarrollan, describen y comparan un gran número de métodos. Se tratan algoritmos para diversos enunciados de problemas, como colas de prioridad, selección y fusión. Algunos de estos algoritmos se utilizan como base de otros algoritmos descritos posteriormente en el libro.

Los métodos de *BÚSQUEDA* para encontrar datos en los archivos son también de gran importancia. Se presentarán métodos avanzados y básicos de búsqueda con árboles y transformaciones de claves digitales, incluyendo árboles de búsqueda binaria, árboles equilibrados, dispersión, árboles de búsqueda digital y tries, así como métodos apropiados para archivos muy grandes. Se presentarán las relaciones entre estos métodos y las similitudes con las técnicas de ordenación.

Los algoritmos de *PROCESAMIENTO DE CADENAS* incluyen una gama de métodos de manipulación de (largas) sucesiones de caracteres. Estos métodos conducen al reconocimiento de patrones en las cadenas, que a su vez conduce al análisis sintáctico. También se desarrollan técnicas para comprimir archivos y para criptografía. Aquí, otra vez, se hace una introducción a temas avanzados mediante el tratamiento de algunos problemas elementales, importantes por sí mismos.

Los *ALGORITMOS GEOMÉTRICOS* son un conjunto de métodos de resolución de problemas a base de puntos y rectas (y de otros objetos geométricos sencillos), que no se han puesto en práctica hasta hace poco tiempo. Se estudian algoritmos para buscar el cerco convexo de un conjunto de puntos, para encontrar intersecciones entre objetos geométricos, para resolver problemas de proximidad y para la búsqueda multidimensional. Muchos de estos métodos complementan de forma elegante las técnicas más elementales de ordenación y búsqueda.

Los *ALGORITMOS SOBRE GRAFOS* son útiles para una variedad de problemas importantes y difíciles. Se desarrolla una estrategia general para la búsqueda en grafos y se aplica a los problemas fundamentales de conectividad, como el camino más corto, el árbol de expansión mínimo, flujo de red y concordancia. Un tratamiento unificado de estos algoritmos demuestra que todos ellos están basados en el mismo procedimiento y que éste depende de una estructura de datos básica desarrollada en una sección anterior.

Los *ALGORITMOS MATEMÁTICOS* presentan métodos fundamentales que proceden del análisis numérico y de la aritmética. Se estudian métodos de la aritmética de enteros, polinomios y matrices, así como también algoritmos para resolver una gama de problemas matemáticos que provienen de muchos contextos: generación de números aleatorios, resolución de sistemas de ecuaciones, ajuste de datos e integración. Se pone énfasis en los aspectos algorítmicos de estos métodos, no en sus fundamentos matemáticos.

Los *TEMAS AVANZADOS* se presentan con el objeto de relacionar el contenido del libro con otros campos de estudio más avanzados. Las computadoras de arquitectura específica, la programación dinámica, la programación lineal, la búsqueda exhaustiva y los problemas NP-completos se examinan desde un punto de vista elemental para dar al lector alguna idea de los interesantes campos de estudio avanzados que sugieren los problemas simples que contiene este libro.

El estudio de los algoritmos es interesante porque se trata de un campo nuevo (casi todos los algoritmos que se presentan en el libro son de hace menos de 25 años), con una rica tradición (algunos algoritmos se conocen desde hace miles de años). Se están haciendo constantemente nuevos descubrimientos y pocos algoritmos se entienden por completo. En este libro se consideran tanto algoritmos difíciles, complicados y enredados, como algoritmos fáciles, simples y elegantes. El desafío consiste en comprender los primeros y apreciar los últimos en el marco de las diferentes aplicaciones posibles. Al hacerlo se descubrirá una variedad de herramientas eficaces y se desarrollará una forma de «pensamiento algorítmico» que será muy útil para los desafíos informáticos del porvenir.

C++ (y C)

A lo largo de este libro se va a utilizar el lenguaje de programación C++. Todos los lenguajes tienen su lado negativo y su lado positivo, y así, la elección de cualquiera de ellos para un libro como éste tiene ventajas e inconvenientes. Pero, como muchos de los lenguajes modernos son similares, si no se utilizan más que algunas instrucciones y se evitan decisiones de realización basadas en las peculiaridades de C++, los programas que se obtengan se podrán traducir fácilmente a otros lenguajes. El objetivo es presentar los algoritmos de la forma más simple y directa que sea posible; C++ permite hacerlo.

Los algoritmos se describen frecuentemente en los libros de texto y en los informes científicos por medio de seudolenguaje —por desgracia esto lleva a menudo a omitir detalles y deja al lector bastante lejos de una implementación práctica—. En este libro se considera que el mejor camino para comprender un algoritmo y comprobar su utilidad es experimentarlo con una situación real. Los lenguajes modernos son lo suficientemente expresivos como para que las implementaciones reales puedan ser tan concisas y elegantes como sus homólogas imaginarias. Se aconseja al lector que se familiarice con el entorno C++ de programación local, ya que en el libro las implementaciones son *programas* pensados para ejecutarlos, experimentar con ellos, modificarlos y *utilizarlos*.

La ventaja de utilizar C++ es que este lenguaje está muy extendido y tiene todas las características básicas que se necesitan en las diversas implementaciones; el inconveniente es que posee propiedades *no* disponibles en algunos otros lenguajes modernos, también muy extendidos, por lo que se deberá tener cuidado y ser consciente de la dependencia que los programas tengan del lenguaje. Algunos de los programas se verán simplificados por las características avanzadas del lenguaje, pero esto ocurre menos veces de las que se podría pensar. Cuando sea apropiado, la presentación de los programas cubrirá los puntos relevantes del lenguaje. En particular, se aprovechará una de las principales virtudes de C++, su compatibilidad con C: el grueso de los códigos se reconocerá fácilmente como C, pero las características importantes de C++ tendrán un papel destacado en muchas de las realizaciones.

Una descripción concisa del lenguaje C++ se encuentra en el libro de Stroustrup *The C++ Programming Language* (segunda edición)¹. El objetivo de este capítulo no es repetir la información de dicho libro, sino más bien ilustrar algunas de las características básicas del lenguaje, por lo que se utilizará como ejemplo la realización de un algoritmo simple (pero clásico). Aparte de la entrada/salida, el código C++ de este capítulo es también código C; también se utilizarán otras características de C++ cuando se consideren estructuras de datos y programas más complicados en algunos de los capítulos siguientes.

Ejemplo: Algoritmo de Euclides

Para comenzar, se considerará un programa en C++ para resolver un problema clásico elemental: «Reducir una fracción determinada a sus términos más elementales». Se desea escribir $2/3$, no $4/6$, $200/300$, o $178468/267702$. Resolver este problema es equivalente a encontrar el *máximo común divisor* (mcd) del numerador y denominador: el mayor entero que divide a ambos. Una fracción se reduce a sus términos más elementales dividiendo el numerador y el denominador por su máximo común divisor. Un método eficaz para encontrar el máximo común divisor fue descubierto por los antiguos griegos hace más de dos mil años: se denomina el *algoritmo de Euclides* porque aparece escrito detalladamente en el famoso tratado *Los elementos*, de Euclides.

El método de Euclides está basado en el hecho de que si u es mayor que v , entonces el máximo común divisor de u y v es el mismo que el de v y $u - v$. Esta observación permite la siguiente implementación en C++:

```
#include <iostream.h>
int mcd(int u, int v)
{
    int t;
    while (u > 0)
    {
        if (u < v) { t = u; u = v; v = t; }
        u = u - v;
    }
    return v;
}
main()
{
```

¹ Existe versión en español de Addison-Wesley/Díaz de Santos (1994) con el título *El lenguaje de programación C++*. (N. del T.)

```

int x, y;
while (cin >> x && cin << y)
    if (x>0 && y>0) cout << x << ' ' << y << ' '
                                << mcd(x,y) << '\n';
}

```

Antes de seguir adelante hay que estudiar las propiedades del lenguaje expuesto en este código. C++ tiene una rigurosa sintaxis de alto nivel que permite identificar fácilmente las principales características del programa. El programa consiste en una lista de funciones, una de las cuales se llama `main()`, y constituye el cuerpo del programa. Las funciones devuelven un valor con la instrucción `return`. C++ incluye una «biblioteca de flujos» para la entrada/salida. La instrucción `include` permite hacer referencia a esta biblioteca. El operador `<<` significa «poner en» el «flujo de salida» `cout`, y, de igual manera, `>>` significa «obtener de» el «flujo de entrada» `cin`. Estos operadores comparan los tipos de datos que se obtienen con los flujos —en este caso se leen como datos de entrada dos enteros, y se obtendrán como salida junto con su máximo común divisor (seguidos por los caracteres `\n` que indican «nueva línea»)—. El valor de `cin >> x` es 0 cuando no hay más datos de entrada.

La estructura del programa anterior es trivial: se leen pares de números de la entrada, y a continuación, si ambos son positivos, se graban en la salida junto con su máximo común divisor. (¿Qué sucede cuando se llama a la función `mcd` con `u` o `v` negativos o con valor cero?) La función `mcd` implementa el algoritmo de Euclides por sí misma: el programa es un bucle que primero se asegura de que $u \geq v$ intercambiando sus valores, si fuera necesario, y reemplazando a continuación `u` por `u - v`. El máximo común divisor de las variables `u` y `v` es siempre igual al máximo común divisor de los valores originales que entraron al procedimiento: tarde o temprano el proceso termina cuando `u` es igual a 0 y `v` es igual al máximo común divisor de los valores originales de `u` y `v` (y de todos los intermedios).

El ejemplo anterior se ha escrito por completo en C++, para que el lector pueda utilizarlo para familiarizarse con algunos sistemas de la programación en C++. El algoritmo de interés se ha escrito como una subrutina (`mcd`), y el programa principal es un «conductor» que utiliza la subrutina. Esta organización es típica, y se ha incluido aquí el ejemplo completo para resaltar que los algoritmos presentados en este libro se entenderán mejor si se implementan y ejecutan con algunos valores de entrada de prueba. Dependiendo de la calidad del entorno de depuración disponible, el lector podría desear llegar más lejos en el análisis de los programas propuestos. Por ejemplo, puede ser interesante ver los valores intermedios que toman `u` y `v` en el bucle `while` del programa anterior.

Aunque el objetivo de esta sección es el lenguaje, no el algoritmo, se debe hacer justicia al clásico algoritmo de Euclides: la implementación anterior puede mejorarse notando que, una vez que $u > v$, se restarán de `u` los múltiples valores de `v` hasta encontrar un número menor que `v`. Pero este número es exac-

tamente el resto que queda al dividir u entre v , que es lo que el operador módulo (%) calcula: el máximo común divisor de u y v es igual al máximo común divisor de v y $u \% v$. Por ejemplo, el máximo común divisor de 461952 y 116298 es 18, tal y como muestra la siguiente secuencia

461952, 116298, 113058, 3240, 2898, 342, 162, 18.

Cada elemento de esta sucesión es el resto que queda al dividir los dos elementos anteriores: la sucesión termina porque 18 es divisor de 162, de manera que 18 es el máximo común divisor de todos los números. Quizás el lector desee modificar la implementación anterior para usar el operador % y comprobar que esta modificación es mucho más eficaz cuando, por ejemplo, se busca el máximo común divisor de un número muy grande y un número muy pequeño. Este algoritmo siempre utiliza un número de pasos relativamente pequeño.

Tipos de datos

La mayor parte de los algoritmos presentados en este libro funcionan con tipos de datos simples: números reales, enteros, caracteres o cadenas de caracteres. Una de las características más importantes de C++ es su capacidad para construir tipos de datos más complejos a partir de estos «ladrillos» elementales. Más adelante se verán muchos ejemplos de esto. Sin embargo, se procurará evitar el uso excesivo de estas facilidades, para no complicar los ejemplos y centrarse en la dinámica de los algoritmos más que en las propiedades de los datos. Se procurará hacerlo sin que esto lleve a una pérdida de generalidad: desde luego, las grandes posibilidades que tiene C++ para realizar construcciones complejas hacen que sea fácil transformar una «maqueta» de algoritmo que opera sobre tipos de datos sencillos en una versión de «tamaño natural» que realiza una operación crítica de una clase de C++. Cuando los métodos básicos se expliquen mejor en términos de tipos definidos por el usuario, así se hará. Por ejemplo, los métodos geométricos de los Capítulos 24-28 están basados en modelos para puntos, líneas, polígonos, etc.; y los métodos de colas de prioridad del Capítulo 11 y los métodos de búsqueda de los Capítulos 14-18 se expresan mejor como conjuntos de operaciones asociados a estructuras de datos particulares, utilizando el constructor clase de C++. Se volverá a este punto en el Capítulo 3, y se verán otros muchos ejemplos a lo largo del libro.

Algunas veces, la conveniente representación de datos a bajo nivel es la clave del rendimiento. Teóricamente, la forma de realizar un programa no debería depender de cómo se representan los números o de cómo se codifican los caracteres (por escoger dos ejemplos), pero el precio que hay que pagar para conseguir este ideal es a veces demasiado alto. Ante este hecho, en el pasado los programadores optaron por la drástica postura de irse al lenguaje ensamblador o al lenguaje máquina, donde hay pocas limitaciones para la representación. Afor-

tunadamente, los lenguajes modernos de alto nivel ofrecen mecanismos para crear representaciones razonables sin llegar a tales extremos. Esto permite justificar algunos algoritmos clásicos importantes. Por supuesto, tales mecanismos dependen de cada máquina, y no se estudian aquí con mucho detalle, excepto para indicar cuándo son apropiados. Este punto se tratará más detalladamente en los Capítulos 10, 17 y 22, al examinar los algoritmos basados en representaciones binarias de datos.

También se tratará de evitar el uso de representaciones que dependen de la máquina, al considerar algoritmos que operan sobre caracteres y cadenas de caracteres. Con frecuencia, se simplifican los ejemplos para trabajar únicamente con las letras mayúsculas de la A a la Z, utilizando un sencillo código en el que la i -ésima letra del alfabeto está representada por el entero i . La representación de caracteres y de cadenas de caracteres es una parte tan fundamental de la interfaz entre el programador, el lenguaje de programación y la máquina, que se debería estar seguro de que se entiende totalmente antes de implementar algoritmos que procesen tales datos —en este libro se dan métodos basados en representaciones sencillas que, por lo tanto, son fáciles de adaptar—.

Se utilizarán números enteros (`int`) siempre que sea posible. Los programas que utilizan números de coma flotante [†] (`float`) pertenecen al dominio del *análisis numérico*. Por lo regular, su utilización va íntimamente ligada a las propiedades matemáticas de la representación. Se volverá sobre este punto en los Capítulos 37, 38, 39, 41 y 43, donde se presentan algunos algoritmos numéricos fundamentales. Mientras tanto, se limitan los ejemplos a la utilización de los números enteros, incluso cuando los números reales puedan parecer más apropiados, para evitar la ineficacia e inexactitud que suelen asociarse a las representaciones mediante números de coma flotante.

Entrada/Salida

Otro dominio en el que la dependencia de la máquina es importante es la interacción entre el programa y sus datos, que normalmente se designa como *entrada/salida*. En los sistemas operativos este término se refiere al intercambio de datos entre la computadora y los soportes físicos tales como un disco o una cinta magnética; se hablará sobre tales materias únicamente en los Capítulos 13 y 18. La mayor parte de las veces se busca un medio sistemático para obtener datos y enviar los resultados a las implementaciones de algoritmos, tales como la función `mcd` anterior.

Cuando se necesite «leer» y «escribir», se utilizarán las características normales de C++, invocando lo menos posible algunos formatos extra disponibles.

[†] En realidad, al ejecutar un programa se debe usar el punto decimal en lugar de la coma para evitar errores durante la ejecución. Asimismo, al escribir cifras, se recomienda no usar puntos para separar los millares o millones. (N. del E.)

Nuevamente, el objetivo es mantener programas concisos, manejables y fácilmente traducibles; una razón por la que el lector podría desear modificar los programas es para mejorar su interfaz con el programador. Pocos, si existe alguno, de los entornos de programación modernos como C++ toman `cin` o `cout` como referencia al medio externo; en su lugar se refieren a «dispositivos lógicos» o a «flujos» de datos. Así, la salida de un programa puede usarse como la entrada de otro, sin ninguna lectura o escritura física. La tendencia a hacer flujos de entrada/salida en las implementaciones de este libro las hace más útiles en tales entornos.

En realidad, en muchos entornos modernos de programación son apropiadas y fáciles de utilizar las representaciones gráficas como las utilizadas en las figuras de este libro. Como se precisa en el epílogo, estas figuras realmente se generan por los propios programas, lo que ha llevado a una mejora sustancial de la interfaz.

Muchos de los métodos que se presentan son apropiados para utilizarlos dentro de grandes sistemas de aplicaciones, de manera que la mejor forma de suministrar datos es mediante el uso de parámetros. Éste es el método utilizado por el procedimiento `mcd` visto anteriormente. También algunas de las implementaciones de capítulos posteriores del libro usarán programas de capítulos anteriores. De nuevo, para evitar desviar la atención de los algoritmos en sí mismos, se resistirá a la tentación de «empaquetar» las implementaciones para utilizarlas como programas de utilidad general. Seguramente, muchas de las implementaciones que se estudiarán son bastante apropiadas como punto de partida para tales utilidades, pero se planteará un gran número de preguntas acerca de la dependencia del sistema o de la máquina, cuyas respuestas se silenciarán aquí, pero que pueden obtenerse de forma satisfactoria durante el desarrollo de tales paquetes.

Algunas veces, se escribirán programas para operar con datos «globales», para evitar una parametrización excesiva. Por ejemplo, la función `mcd` podría operar directamente con `x` e `y`, sin necesidad de recurrir a los parámetros `u` y `v`. Esto no está justificado en este caso porque `mcd` es una función bien definida en términos de sus dos entradas, pero cuando varios algoritmos operan sobre los mismos datos, o cuando se pasa una gran cantidad de datos, se podrían utilizar variables globales para reducir la expresión algorítmica y para evitar mover datos innecesariamente. Por otra parte, C++ es un lenguaje ideal para «encapsular» los algoritmos y sus estructuras de datos asociadas para hacer explícitas las interfaces, y se tenderá a usar datos globales muchas menos veces en las implementaciones en C++ que en los correspondientes programas en C o Pascal.

Comentarios finales

En *The C++ Programming Language* y en los capítulos que siguen se muestran otros muchos ejemplos parecidos al programa anterior. Se invita al lector a ho-

pear el manual, implementar y probar algunos programas sencillos y posteriormente leer el manual con detenimiento para familiarizarse con las características básicas de C++.

Los programas en C++ que se muestran en este libro deben servir como descripciones precisas de los algoritmos, como ejemplos de implementaciones completas y como punto de partida para la realización de programas prácticos. Como se ha mencionado anteriormente, los lectores experimentados en otros lenguajes no deben tener dificultades para leer los algoritmos presentados en C++ e implementarlos en otros lenguajes. Por ejemplo, la siguiente es una implementación en Pascal del algoritmo de Euclides:

```
program euclides(input, output);
var x, y: integer;
function mcd(u, v: integer): integer;
  var t: integer;
begin
repeat
  if u < v then
    begin t:=u; u:=v; v:=t end;
    u:=u-v
  until u=0;
  mcd:=v
end;
begin
while not eof do
  begin
    readln(x, y);
    if (x>0) and (y>0) then writeln(x, y, mcd(x, y))
  end;
end.
```

En este algoritmo hay una correspondencia prácticamente exacta entre las sentencias en C++ y Pascal, como era la intención; sin embargo, no es difícil desarrollar implementaciones más precisas en ambos lenguajes. En este caso la realización en C++ se diferencia de la realizada en C únicamente en la entrada/salida: el objetivo será mantener esta compatibilidad siempre que sea natural hacerlo, aunque, por supuesto, la mayoría de los programas de este libro utilizan instrucciones C++ que no están disponibles en C.

Ejercicios

1. Implementar la versión clásica del algoritmo de Euclides presentado en el texto.

2. Comprobar qué valores de $u \% v$ calcula el sistema en C++ cuando u y v no son siempre positivos.
3. Implementar un procedimiento para hacer irreducible una fracción dada, utilizando una struct `fraccion { int numerador; int denominador; }.`
4. Escribir una función `int convertir ()` que lea un número decimal cifra a cifra, termine cuando encuentre un espacio en blanco y devuelva el valor del número.
5. Escribir una función `binario (int x)` que presente el equivalente binario de un número.
6. Obtener los valores que toman u y v cuando se invoca la función `mcd` con la llamada inicial `mcd(12345, 56789)`.
7. ¿Cuántas instrucciones de C++ se ejecutan exactamente en la llamada del ejercicio anterior?
8. Escribir un programa que calcule el máximo común divisor de *tres* enteros u , v y w .
9. Encontrar el mayor par de números representables como enteros en el sistema C++, cuyo máximo común divisor sea 1.
10. Implementar el algoritmo de Euclides en FORTRAN y BASIC.

Estructuras de datos elementales

En este capítulo se presentan los métodos básicos de organizar los datos para procesarlos mediante programas de computadora. En muchas aplicaciones la decisión más importante en la implementación es elegir la estructura de datos adecuada: una vez realizada la elección, lo único que se necesitan son algoritmos simples. Para los mismos datos, algunas estructuras requieren más o menos espacio que otras; para las mismas operaciones con datos, algunas estructuras requieren un número distinto de algoritmos, unos más eficaces que otros. Esto ocurrirá con frecuencia a lo largo de este libro, porque la elección del algoritmo y de la estructura de datos está estrechamente relacionada y continuamente se buscan formas de ahorrar tiempo o espacio mediante una elección adecuada.

Una estructura de datos no es un objeto pasivo: es preciso considerar también las operaciones que se ejecutan sobre ella (y los algoritmos empleados en estas operaciones). Este concepto se formaliza en la noción de *tipo de datos abstracto*, que se analiza al final del capítulo. Pero como el mayor interés está en las implementaciones concretas, se fijará la atención en las manipulaciones y representaciones específicas.

Se trabajará con arrays, listas enlazadas, pilas, colas y otras variantes sencillas. Éstas son estructuras de datos clásicas con un gran número de aplicaciones: junto con los árboles (ver Capítulo 4), forman prácticamente la base de todos los algoritmos que se consideran en este libro. En este capítulo se verán las representaciones básicas y los métodos de manipulación de estas estructuras, se trabajará con algunos ejemplos concretos de utilización y se presentarán puntos específicos, como la administración del almacenamiento.

Arrays

Tal vez el *array* sea la estructura de datos más importante, que se define como una primitiva tanto en C++ como en otros muchos lenguajes de programación. Un array es un número fijo de elementos de datos que se almacenan de forma contigua y a los que se accede por un índice. Se hace referencia al *i*-ésimo elemento de un array a como *a[i]*. Es responsabilidad del programador almacenar en una posición *a[i]* de un array un valor coherente antes de llamarlo; descuidar esto es uno de los errores más comunes de la programación.

Un sencillo ejemplo de la utilización de un array es el siguiente programa, que imprime todos los números primos menores de 1.000. El método utilizado, que data del siglo III a.C., se denomina la «criba de Eratóstenes»:

```
const int N = 1000;
main()
{
    int i, j, a[N+1];
    for (a[1] = 0, i = 2; i <= N; i++) a[i] = 1;
    for (i = 2; i <= N/2; i++)
        for (j = 2; j <= N/i; j++)
            a[i*j] = 0;
    for (i = 1; i <= N; i++)
        if (a[i]) cout << i << ' ';
    cout << '\n';
}
```

Este programa emplea un array constituido por el tipo más sencillo de elementos, los valores booleanos (0-1). El objetivo del mismo es poner en *a[i]* el valor 1 si *i* es un número primo, o poner un 0 si no lo es. Para todo *i*, se pone a 0 el elemento del array que corresponde a cualquier múltiplo de *i*, ya que cualquier número que sea múltiplo de cualquier otro número no puede ser primo. A continuación se recorre el array una vez más, imprimiendo los números primos. Primero se «inicializa» el array para indicar los números que se sabe que no son primos: el algoritmo pone a 0 los elementos del array que corresponden a índices conocidos como no primos. Se puede mejorar la eficacia del programa, comprobando *a[i]* antes del *for* del bucle que involucra a *j*, ya que si *i* no es primo, los elementos del array que corresponden a todos sus múltiplos deben haberse marcado ya. Podría hacerse un empleo más eficaz del espacio mediante el uso explícito de un array de bits y no de enteros.

La criba de Eratóstenes es uno de los algoritmos típicos que aprovechan la posibilidad de acceder directamente a cualquier elemento de un array. El algoritmo accede a los elementos del array secuencialmente, uno detrás de otro. En muchas aplicaciones, es importante el orden secuencial; en otras se utiliza por-

que es tan bueno como cualquier otro. Pero la característica principal de los arrays es que *si se conoce el índice*, se puede acceder a cualquier elemento en un tiempo constante.

El tamaño de un array debe conocerse de antemano: para ejecutar el programa anterior para un valor diferente de N, es necesario cambiar la constante N y después volver a compilar y a ejecutar. En algunos entornos de programación, es posible declarar el tamaño de un array durante la ejecución (de modo que se podría conseguir, por ejemplo, que un usuario introduzca el valor de N para obtener los números primos menores que N sin el desperdicio de memoria provocado al definir un tamaño del array tan grande como el valor máximo que se permita teclear al usuario). En C++ es posible lograr este efecto mediante la apropiada utilización del mecanismo de asignación de la memoria, pero sigue siendo una propiedad fundamental de los arrays que sus tamaños sean fijos y se deban conocer antes de utilizarlos.

Los arrays son estructuras de datos fundamentales que tienen una correspondencia directa con los sistemas de administración de memoria, en prácticamente todas las computadoras. Para poder recuperar el contenido de una palabra de la memoria es preciso proporcionar una dirección en lenguaje máquina. Así se podría representar la memoria total de la computadora como si fuera un array, en el que las direcciones de memoria correspondieran a los índices del mismo. La mayor parte de los procesadores de lenguaje, al traducir programas a lenguaje máquina, construyen arrays bastante eficaces que permiten acceder directamente a la memoria.

Otra forma normal de estructurar la información consiste en utilizar una tabla de números organizada en filas y columnas. Por ejemplo, una tabla de las notas de los estudiantes de un curso podría tener una fila para cada estudiante, y una columna para cada asignatura. En una computadora, esta tabla se representaría como un *array bidimensional* con dos índices, uno para las filas y otro para las columnas. Hay varios algoritmos que son inmediatos para manejar estas estructuras: por ejemplo, para calcular la nota media de una asignatura, se suman todos los elementos de una columna y se dividen por el número de filas; para calcular la nota media del curso de un estudiante en particular, se suman todos los elementos de una fila y se dividen por el número de columnas. Los arrays bidimensionales se utilizan generalmente en aplicaciones de este tipo. En una computadora se utilizan a menudo más de dos dimensiones: un profesor podría utilizar un tercer índice para mantener en tablas las notas de los estudiantes de una serie de años.

Los arrays también se corresponden directamente con los *vectores*, término matemático utilizado para las listas indexadas de objetos. Análogamente, los arrays bidimensionales se corresponden con las *matrices*. Los algoritmos para procesar estos objetos matemáticos se estudian en los Capítulos 36 y 37.

Listas enlazadas

La segunda estructura de datos elementales a considerar es la *lista enlazada*, que se define como una primitiva en algunos lenguajes de programación (concretamente en Lisp) pero no en C++. Sin embargo, C++ proporciona operaciones básicas que facilitan el uso de listas enlazadas.

La ventaja fundamental de las listas enlazadas sobre los arrays es que su tamaño puede aumentar y disminuir a lo largo de su vida. En particular, no se necesita conocer de antemano su tamaño máximo. En aplicaciones prácticas, esto hace posible que frecuentemente se tengan varias estructuras de datos que comparten el mismo espacio, sin tener que prestar en ningún momento una atención particular a su tamaño relativo.

Una segunda ventaja de las listas enlazadas es que proporcionan flexibilidad, lo que permite que los elementos se reordenen eficazmente. Esta flexibilidad se gana en detrimento de la rapidez de acceso a cualquier elemento de la lista. Esto se verá más adelante, después de que se hayan examinado algunas de las propiedades básicas de las listas enlazadas y algunas de las operaciones fundamentales que se llevan a cabo con ellas.

Una lista enlazada es un conjunto de elementos organizados secuencialmente, igual que un array. Pero en un array la organización secuencial se proporciona implícitamente (por la posición en el array), mientras que en una lista enlazada se utiliza un orden explícito en el que cada elemento es parte de un «nodo» que contiene además un «enlace» con el nodo siguiente. La Figura 3.1 muestra una lista enlazada, con los elementos representados por letras, los nodos por círculos y los enlaces por líneas que conectan los nodos. Más adelante se verá, de forma detallada, cómo se representan las listas en la computadora; por ahora se hablará simplemente de nodos y enlaces.

Incluso la sencilla representación de la Figura 3.1 pone en evidencia dos detalles que deben considerarse. Primero, todo nodo tiene un enlace, por lo que el enlace del último nodo de la lista debe designar a algún nodo «siguiente». Con este fin se adopta el convenio de tener un nodo «ficticio», que se denomina *Z*: el último nodo de la lista apuntará a *Z* y *Z* se apuntará a sí mismo. En segundo lugar, también por convenio, se tendrá un nodo ficticio en el otro extremo de la lista, que se denomina *cabeza*, y que apuntará al primer nodo de la lista. El principal objetivo de los nodos ficticios es que resulte más cómodo hacer ciertas manipulaciones con los enlaces, especialmente con aquellos que están relacionados con el primer y último nodo de la lista. Más adelante, se ve-



Figura 3.1 Una lista enlazada.

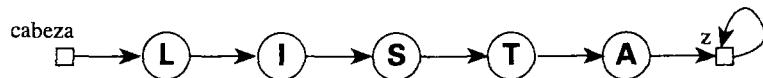


Figura 3.2 Una lista enlazada con sus nodos ficticios.

rán más normas que se toman por convenio. La Figura 3.2 muestra la estructura de la lista con estos nodos ficticios.

Esta representación explícita de la ordenación permite que ciertas operaciones se ejecuten mucho más eficazmente de lo que sería posible con arrays. Por ejemplo, suponiendo que se quiere mover la A desde el final de la lista al principio, en un array se tendría que mover cada elemento para hacer sitio en el comienzo para el nuevo elemento; en una lista enlazada, simplemente se cambian tres enlaces, como se muestra en la Figura 3.3. Las dos versiones que aparecen en la Figura 3.3 son equivalentes; simplemente están dibujadas de manera diferente. Se hace que el nodo que contiene a A apunte a L, que el nodo que contiene a T apunte a z, y que cabeza apunte a A. Aun cuando la lista fuese muy larga, se podría hacer este cambio estructural modificando solamente tres enlaces.

La operación siguiente, que es antinatural e inconveniente en un array, es todavía más importante. Se trata de «insertar» un elemento en una lista enlazada (lo que hace aumentar su longitud en una unidad). La Figura 3.4 muestra cómo insertar X en la lista del ejemplo, poniendo X en un nodo que apunte a T, y a continuación haciendo que el nodo que contiene a S apunte al nuevo nodo. En esta operación sólo se necesita cambiar dos enlaces, cualquiera que sea la longitud de la lista.

De igual forma, se puede hablar de «eliminar» un elemento de una lista enlazada (lo que hace disminuir su longitud en una unidad). Por ejemplo, la tercera lista de la Figura 3.4 muestra cómo eliminar X de la segunda lista haciendo simplemente que el nodo que contiene a S apunte a T, saltándose a X. Ahora,

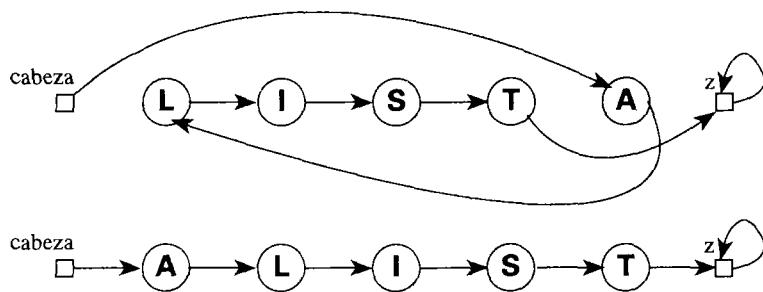


Figura 3.3 Reordenación de una lista enlazada.

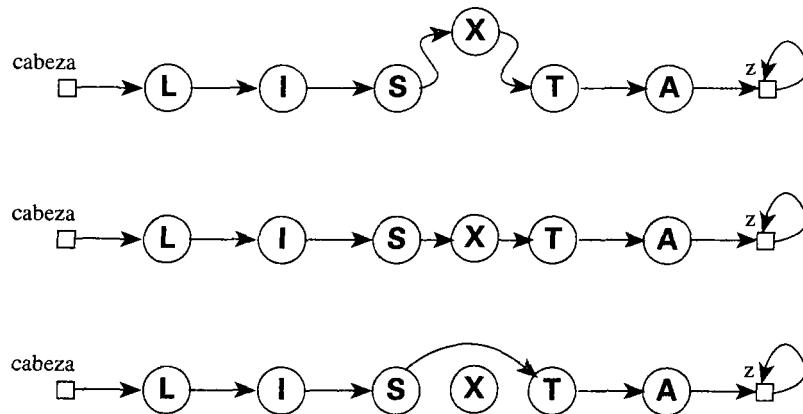


Figura 3.4. Inserción y borrado en una lista completa.

el nodo que contiene a X todavía existe (de hecho, todavía apunta a T), y quizás debería eliminarse de alguna manera; pero el hecho es que X no forma parte de la lista, y no puede accederse a él mediante enlaces desde cabeza. Se volverá sobre este punto más adelante.

Por otra parte, hay otras operaciones para las que las listas enlazadas *no son* apropiadas. La más obvia de estas operaciones es «encontrar el k-ésimo elemento» (encontrar un elemento dado su índice): en un array esto se hace fácilmente accediendo a $a[k]$, pero en una lista hay que moverse a lo largo de k enlaces. Otra operación que es antinatural en las listas enlazadas es «encontrar el elemento anterior a uno dado». Si el único dato de la lista del ejemplo es el enlace a T, entonces la única forma de poder encontrar el enlace a T es comenzar en *cabeza* y recorrer la lista para encontrar el nodo que apunta a A. En realidad, esta operación es necesaria si se desea eliminar un nodo concreto de una lista enlazada, ya que ¿de qué otra manera se encontrará el nodo cuyo enlace debe modificarse? En muchas aplicaciones se puede rodear este problema transformando la operación «eliminar» en la «eliminar el nodo siguiente». En la inserción se puede evitar un problema similar haciendo que la operación sea «insertar un elemento dado *detrás de* un nodo determinado» de la lista.

Para ilustrar cómo podría implementarse en C++ una lista enlazada básica, se comenzará especificando con precisión el formato de los nodos de la lista y construyendo una lista vacía, como se indica a continuación:

```
struct nodo
{ int clave; struct nodo *siguiente; };
struct nodo *cabeza, *z;
```

```
cabeza = new nodo; z = new nodo;  
cabeza->siguiente = z; z->siguiente = z;
```

La declaración struct indica que las listas están compuestas por nodos y que cada nodo contiene un número entero y un puntero al siguiente nodo de la lista. La variable clave es un entero, únicamente para simplificar el programa; pero podría ser de cualquier tipo —el puntero siguiente es la clave de la lista—. El asterisco indica que las variables cabeza y z se declaran como punteros a los nodos. En realidad éstos se crean únicamente cuando se llama a la función integrada new. Esto oculta un complejo mecanismo que tiene como finalidad aliviar al programador de la carga de asignar «memoria» para los nodos según va creciendo la lista. Más adelante se estudiará este mecanismo con mayor detalle. La notación «flecha» (un signo menor seguido de un signo mayor) se utiliza en C++ para seguir a los punteros a través de las estructuras. Se escribe una referencia a un enlace seguida por este símbolo para indicar una referencia al nodo al que apunta ese enlace. Así, el código anterior crea dos nuevos nodos referenciados por cabeza y z y pone a ambos apuntando a z.

Para insertar en una lista enlazada detrás de un nodo dato t un nuevo nodo con el valor de la clave v, se crea el nodo (x = new nodo) y se pone en el valor clave (x->clave = v), después se copia en el enlace de t (x->siguiente = t->siguiente) y se hace que el enlace de t apunte al nuevo nodo (t->siguiente = x).

Para extraer de una lista enlazada el nodo siguiente a un nodo dado t, se obtiene un puntero a ese nodo (x = t->siguiente), se copia el puntero en t para sacarlo de la lista (t->siguiente = x->siguiente) y devolverlo al sistema de asignación de memoria empleando el procedimiento integrado delete, a menos que la lista estuviese vacía (if (x!=z) delete x).

Se invita al lector a que compare estas implementaciones en C++ con las presentadas en la Figura 3.4. Es interesante destacar que el nodo cabeza evita el tener que hacer una comprobación especial en la inserción de un elemento al principio de la lista, y el nodo z proporciona una forma apropiada de comprobar la eliminación de un elemento en una lista vacía. Se verá otra utilización de z en el Capítulo 14.

En capítulos posteriores se verán muchos ejemplos de aplicaciones de este tipo y otras operaciones básicas sobre listas enlazadas. Como las operaciones sólo se componen de unas cuentas instrucciones, con frecuencia se manipularán las listas directamente en lugar de hacerlo mediante los tipos de datos. Como ejemplo, se considera el siguiente programa para resolver el denominado «problema de Josefo» en la misma línea de la criba de Eratóstenes. Se supone que N personas han decidido cometer un suicidio masivo, disponiéndose en un círculo y matando a la M -ésima persona alrededor del círculo, cerrando las filas a medida que cada persona va abandonando el círculo. El problema consiste en averiguar qué persona será la última en morir (¡aunque quizás al final cambie de idea!), o, más generalmente, encontrar el orden en que mueren las personas. Por ejem-

plo, si $N = 9$ y $M = 5$, las personas morirán en el orden 5 1 7 4 3 6 9 2 8. El siguiente programa lee N y M y obtiene este orden:

```
struct nodo
{ int clave; struct nodo *siguiente; };
main()
{
    int i, N, M;
    struct nodo *t, *x;
    cin >> N >> M;
    t = new nodo; t->clave = 1; x = t;
    for (i = 2; i <= N; i++)
    {
        t->siguiente = new nodo;
        t = t->siguiente; t->clave = i;
    }
    t->siguiente = x;
    while (t != t->siguiente)
    {
        for (i = 1; i < M; i++) t = t->siguiente;
        cout << t->siguiente; t->siguiente = x->siguiente;
        delete x;
    }
    cout << t->clave << '\n';
}
```

El programa utiliza una lista enlazada «circular» para simular directamente la secuencia de ejecuciones. Primero, se construye la lista para las claves desde 1 a N de forma que la variable x ocupe el principio de la lista en el momento de su creación, después el puntero del último nodo de la lista se pone en x . El programa continúa recorriendo la lista, contando hasta el elemento $M - 1$ y eliminando el siguiente, hasta que se deje uno sólo (que entonces se apunta a sí mismo). Se observa que se llama a `delete` para suprimir elementos, lo que corresponde a una ejecución: éste es el operador opuesto a `new`, como se mencionó anteriormente.

Las listas circulares se emplean a veces como una alternativa a la utilización de los nodos ficticios `cabeza` o `z`, con un nodo ficticio para marcar el principio (y el final) de la lista y como ayuda en el caso de las listas vacías.

La operación «encontrar el elemento anterior a uno dado» se puede realizar mediante la utilización de una *lista doblemente enlazada*, en la que se mantienen dos enlaces para cada nodo, uno para el elemento anterior, y otro para el elemento posterior. El coste de contar con esta capacidad extra es duplicar el número de enlaces manipulados por cada operación básica; de manera que no

es normal que se utilicen, a menos que se requiera específicamente. Por otro lado, como se mencionó antes, si se va a eliminar un nodo y sólo se dispone de un enlace al mismo (que quizás también es parte de alguna otra estructura de datos), pueden utilizarse enlaces dobles.

Asignación de memoria

Como se mostró anteriormente, los punteros de C++ proporcionan una manera adecuada de implementar listas; pero existen otras alternativas. En esta sección se verá cómo se utilizan los arrays para implementar listas enlazadas así como la relación entre esta técnica y la representación real de las listas en un programa en C++. Como ya se mencionó, los arrays son una representación bastante directa de la memoria de la computadora, por lo que el análisis de cómo se implementa una estructura de datos de este tipo proporcionará algún conocimiento sobre cómo podría representarse esta estructura a bajo nivel de la computadora. En particular, interesa ver cómo podrían representarse al mismo tiempo varias listas.

Para representar directamente listas enlazadas mediante arrays, se utilizan índices en lugar de enlaces. Una manera de proceder sería definir un array de registros parecidos a los anteriores, pero utilizando enteros (`int`) para los índices del array, en lugar de punteros al campo `siguiente`. Una alternativa, que suele resultar más conveniente, es utilizar «arrays paralelos»: se guardan los elementos en un array `clave` y los enlaces en otro array `siguiente`. Así `clave[siguiente[cabeza]]` se refiere a la información asociada con el primer elemento de la lista, `clave[siguiente[siguiente[cabeza]]]` con el segundo, y así sucesivamente. La ventaja de utilizar arrays paralelos es que la estructura puede construirse «sobre» los datos: el array `clave` contiene datos y sólo datos; toda la estructura está en el array paralelo `siguiente`. Por ejemplo, se puede construir otra lista empleando el mismo array de datos y un paralelo de «enlace» diferente, o se pueden añadir más datos con más arrays paralelos.

La siguiente línea de código implementa la operación «insertar después de» en una lista enlazada representada por los arrays paralelos `clave` y `siguiente`

```
clave[x] = v; siguiente[x] = siguiente[t]; siguiente[t] = x++;
```

El «puntero» `x` sigue la pista de la siguiente posición que está sin ocupar en el array, de manera que no se necesita llamar a la función de asignación de memoria `new`. Para extraer un nodo se escribe `siguiente[t] = siguiente[siguiente[t]]`, pero se pierde la posición del array «a la que apunta» `siguiente[t]`. Más adelante, se verá cómo podría recuperarse este espacio perdido.

La Figura 3.5 muestra cómo se podría representar la lista del ejemplo mediante arrays paralelos y cómo se relaciona esta representación con la represen-

tación gráfica que se ha estado utilizando. Los arrays clave y siguiente se muestran en el primer diagrama de la izquierda, como aparecerían si se insertara T I L S A en una lista inicialmente vacía, con T, I y L insertados después de cabeza, S después de I y A después de T. La posición 0 es cabeza y la posición 1 es z (se ponen al inicializar la lista) de manera que como siguiente[0] es 4, el primer elemento de la lista es clave[4] (L); como siguiente[4] es 3, el segundo elemento de la lista es clave[3] (I), etc. En el segundo diagrama por la izquierda, los índices para el array siguiente se reemplazan por líneas y en lugar de poner un 4 en siguiente[0], se dibuja una línea desde el nodo 0 hasta el nodo 4, etc. En el tercer diagrama se desenredan los enlaces para ordenar los elementos de la lista, uno a continuación de otro. Y para concluir, a la derecha aparece la lista en la representación gráfica habitual.

Lo esencial del caso es considerar cómo podrían implementarse los procedimientos integrados `new` y `delete`. Se supone que el único espacio disponible para nodos y enlaces son los arrays anteriores; esta presunción lleva a la situación en la que se encuentra el sistema cuando tiene que permitir que se aumente o disminuya el tamaño de una estructura de datos a partir de una estructura fija (la memoria). Por ejemplo, suponiendo que el nodo que contiene a L se debe eliminar del ejemplo de la Figura 3.5, es fácil reordenar los enlaces de manera que el nodo no esté mucho tiempo enganchado a la lista, pero ¿qué hacer con el espacio ocupado por ese nodo?, y ¿cómo encontrar espacio para un nodo cuando se llame a la función `new` y se necesite más espacio?

Reflexionando se ve que la solución está clara: ¡es suficiente con utilizar otra lista enlazada para seguir la pista del espacio libre!, denominándola como la «lista libre». Entonces, al eliminar (`delete`) un nodo de la primera lista, se inserta en la lista libre y cuando se necesite un nodo `new`, se obtiene *eliminándolo* de la

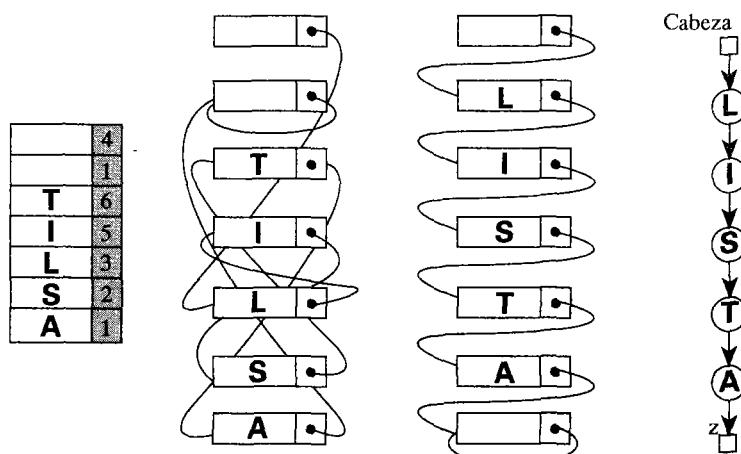


Figura 3.5 Implementación de una lista enlazada mediante un array.

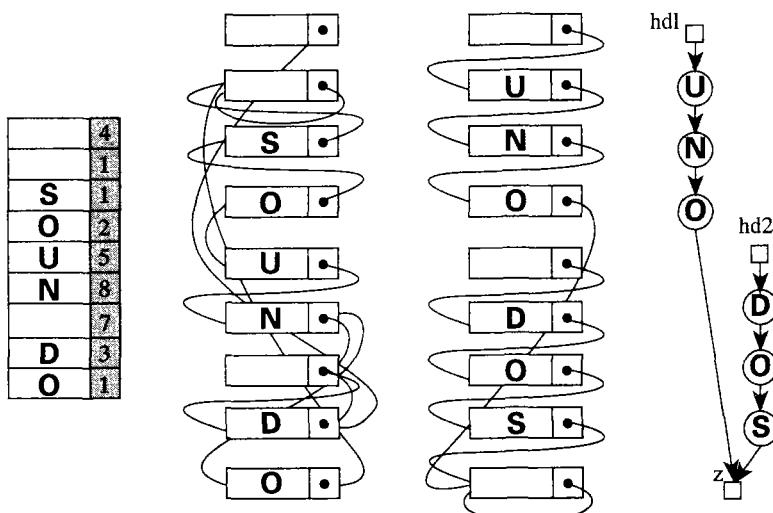


Figura 3.6 Dos listas compartiendo el mismo espacio.

lista libre. Este mecanismo permite tener varias listas diferentes ocupando el mismo array.

En la Figura 3.6 se muestra un ejemplo sencillo con dos listas (pero sin lista libre). Hay dos nodos cabeza de lista $hd1^* = 0$ y $hd2 = 6$, pero ambas listas pueden compartir el mismo z . Una implementación típica de C++ que utilice la construcción `class` tendría un nodo cabeza y un nodo cola asociado a cada lista. Ahora, `siguiente[0]` es 4, y por tanto el primer elemento de la primera lista es `clave[4]` (N); como `siguiente[6]` es 7, el primer elemento de la segunda lista es `clave[7]` (D), etc. Los otros diagramas de la Figura 3.6 muestran el resultado de remplazar los valores `siguiente` por líneas, desenredando los nodos y cambiando la representación gráfica simple, como en la Figura 3.5. Esta misma técnica podría utilizarse para mantener varias listas en el mismo array, una de las cuales debería ser una lista libre, como se describió anteriormente.

Cuando el sistema dispone de un gestor de memoria, como ocurre en C++, no hay razón para suplantarla de esta manera. La descripción anterior se realiza para indicar cómo hace el sistema la gestión de la memoria. (Si se trabaja con un sistema que no asigna memoria, la descripción anterior proporciona un buen punto de partida para una implementación.) En la práctica, el problema que se acaba de ver es mucho más complejo, ya que no todos los nodos son necesariamente del mismo tamaño. Además algunos sistemas relevan al usuario de la necesidad de eliminar explícitamente los nodos, mediante el uso de algoritmos de «recolección de basura», que eliminan de forma automática los nodos que no estén referenciados por ningún enlace. Para resolver estas dos situaciones se ha

* Abreviatura de `head1` (cabeza1). (N. del T.)

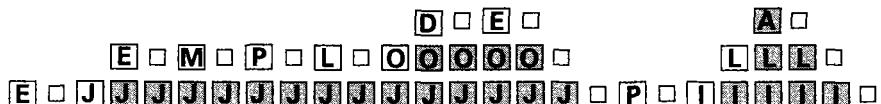


Figura 3.7 Características dinámicas de una pila.

desarrollado un buen número de complicados algoritmos de asignación de memoria.

Pilas

Hasta aquí se ha considerado la forma de estructurar los datos con el fin de insertar, eliminar o acceder arbitrariamente a los distintos elementos. En realidad, resulta que para muchas aplicaciones es suficiente con considerar varias restricciones (más bien fuertes) sobre la forma de acceder a la estructura de datos. Tales restricciones son beneficiosas por dos motivos: primero, porque pueden aliviar la necesidad que tiene el programa de utilizar la estructura de datos para analizar ciertos detalles (por ejemplo, memorizando los enlaces o los índices de los elementos); segundo, porque permiten implementaciones más simples y flexibles, tendiendo a limitar el número de operaciones.

La estructura de datos de acceso restrictivo más importante es la *pila*, en la que sólo existen dos operaciones básicas: se puede meter un elemento en la pila (insertarlo al principio) y se puede sacar un elemento (eliminarlo del principio). Una pila funciona de forma parecida a la bandeja de «entradas» de un ejecutivo muy ocupado: el trabajo se amontona en la pila y cuando el ejecutivo está preparado para hacer alguna tarea, coge la de la parte más alta del montón. Pudiera ser que algún documento se quedara bloqueado en el fondo de la pila durante algún tiempo, pero un buen ejecutivo se las arregla para conseguir vaciar la pila periódicamente. A veces los programas de computadora se organizan naturalmente de esta manera, posponiendo algunas tareas mientras se hacen otras, y por ello las pilas representan una estructura de datos fundamental para muchos algoritmos.

La Figura 3.7 muestra un ejemplo de una pila desarrollada a través de una serie de operaciones meter y sacar, representadas por la secuencia:

*E * JE * M * P * L * QD * E *** P * ILA ****

Cada letra de esta lista significa «meter» (la letra) y cada asterisco significa «sa-car».

C++ proporciona una forma excelente para definir y utilizar estructuras de datos fundamentales, tales como las pilas. La siguiente implementación de las operaciones básicas sobre las pilas es el prototipo de muchas implementaciones

que se verán más adelante en el libro. La pila está representada por un array *pila* y un puntero *p* que apunta hacia lo más alto de la pila —las funciones *meter*, *sacar* y *vacía* son implementaciones directas de las operaciones básicas de la pila—. Este código no comprueba si el usuario trata de meter un elemento en una pila llena, o de sacar un elemento de una vacía; aunque la función *vacía* es una forma de comprobarlo posteriormente.

```
class Pila
{
private:
    tipoElemento *pila;
    int p;
public:
    Pila(int max=100)
    { pila = new tipoElemento[max]; p = 0; }
    ~Pila()
    { delete pila; }
    inline void meter(tipoElemento v)
    { pila[p++] = v; }
    inline tipoElemento sacar()
    { return pila[--p]; }
    inline int vacia()
    { return !p; }
};
```

El tipo de elementos contenido en la pila se deja sin especificar con el nombre *tipoElemento*, para tener alguna flexibilidad. Por ejemplo, la sentencia *type-def int tipoElemento;* podría emplearse para tener una pila formada por números enteros. Al final de este capítulo se presentará una alternativa a esto en C++.

El código anterior muestra varias construcciones en C++. La implementación es una *class*, una de las clases del «tipo definido por el usuario» de C++. Con esta definición, una *Pila* tiene la misma condición que *int*, *char* o cualquiera de los otros tipos incorporados. La implementación se divide en dos partes, una parte *private* que especifica cómo están organizados los datos y una parte *public* que define las operaciones permitidas sobre los datos. Los programas que utilizan pilas necesitan referenciar solamente la parte pública sin preocuparse de cómo se implementan las pilas. La función *Pila* es un «constructor» que crea e inicializa la pila y la función *~* es un «destructor» que elimina la pila cuando ya no se le necesita. La palabra clave *inline* indica que se reemplaza la llamada a la función por la función misma, evitando así la sobrecarga de llamadas, que es muy apropiada para funciones cortas como éstas.

En los capítulos siguientes se verán muchas aplicaciones de las pilas: un buen

ejemplo de introducción será examinar la utilización de estas estructuras en la evaluación de expresiones aritméticas. Se supone que se desea encontrar el valor de una expresión aritmética simple formada por multiplicaciones y sumas de enteros, tal como

$$5 * ((9 + 8) * (4 * 6)) + 7.$$

Para realizar este cálculo, hay que guardar aparte algunos resultados intermedios: por ejemplo, si se calcula primero $9+8$, entonces el resultado parcial (17) debe guardarse en algún lugar mientras se calcula $4 * 6$. Una pila constituye el mecanismo ideal para guardar los resultados intermedios de este cálculo.

Para comenzar, se reordena sistemáticamente la expresión de modo que cada operador aparezca después de sus dos argumentos, en lugar de aparecer entre ellos. De manera que al ejemplo anterior le corresponde la expresión

$$5\ 9\ 8\ +\ 4\ 6\ * *\ 7\ +\ *.$$

Esto se denomina *notación polaca inversa* (dado que fue introducida por un célebre lógico polaco), o *postfija*. La forma normal de escribir expresiones aritméticas se denomina *infija*. Una propiedad interesante de la notación postfija es que no se necesitan paréntesis, mientras que en la infija éstos son necesarios para distinguir el orden de las operaciones ya que, por ejemplo, no es lo mismo $5*((9+8)*(4*6))+7$ que $((5*9)+8)*((4*6)+7)$. Una propiedad todavía más interesante de la notación postfija es que proporciona una manera sencilla de ejecutar el cálculo, guardando los resultados intermedios en una pila. El siguiente programa lee una expresión postfija, interpretando cada operando como una orden para «introducir el operando en la pila» y cada operador como una orden para «recuperar los dos operandos de la pila, ejecutar la operación e introducir el resultado».

```
char c; Pila acc(50); int x;
while (cin.get(c))
{
    x = 0;
    while (c == ' ') cin.get(c);
    if (c == '+') x = acc.sacar() + acc.sacar();
    if (c == '*') x = acc.sacar() * acc.sacar();
    while (c>='0' && c<='9')
        { x = 10*x + (c-'0'); cin.get(c); }
    acc.meter(x);
}
cout << acc.sacar() << '\n';
```

La pila guardar se declara y define junto con las otras variables del programa

y las operaciones meter y sacar de guardar se invocan exactamente igual que get para el flujo de entrada cin. Este programa lee cualquier expresión postfija formada por multiplicaciones y sumas de enteros y después obtiene el valor de la expresión. Los espacios en blanco se ignoran y el bucle while convierte a formato numérico los enteros que están en formato alfanumérico para poder realizar los cálculos. En C++ no se especifica el orden en que se ejecutan las dos operaciones sacar (), por lo que se necesitará un código algo más complejo para operadores no commutativos tales como la resta y la división.

El siguiente programa convierte una expresión infija, de paréntesis totalmente permitidos, en una expresión postfija:

```
char c; Pila guardar(50);
while (cin.get(c))
{
    if (c == ')') cout.put(guardar.sacar());
    if (c == '+') guardar.meter(c);
    if (c == '*') guardar.meter(c);
    while (c>='0' && c<='9')
        { cout.put(c); cin.get(c); }
    if (c != '(') cout << ' ';
}
cout << '\n';
```

Los operadores se meten en la pila y los argumentos simplemente pasan a través de ella. Así, los argumentos aparecen en la expresión postfija en el mismo orden que en la expresión infija. Entonces cada paréntesis derecho indica que se obtienen los dos argumentos del último operador, por tanto el propio operador puede retirarse de la pila e imprimirse como salida. Es interesante observar que como sólo se utilizan operadores con exactamente dos operandos, no se necesitan paréntesis izquierdos en la expresión infija (y este programa los omite). Para hacerlo más sencillo, el programa no verifica los errores de la entrada y exige la presencia de espacios entre operadores, paréntesis y operandos.

El paradigma «guardar los resultados intermedios» es fundamental, y así las pilas aparecen frecuentemente. Muchas máquinas llevan a cabo las operaciones básicas de pilas en el hardware, al implementar de manera natural mecanismos de llamada a funciones, como, por ejemplo, guardar el entorno actual en la entrada de un procedimiento introduciendo información en una pila, restaurar el entorno en la salida recuperando la información de la pila, etc. Algunas calculadoras y lenguajes basan sus métodos de cálculo en operaciones con pilas: cada operación obtiene sus argumentos de la pila y devuelve sus resultados a la misma. Como se verá en el Capítulo 5, las pilas aparecen con frecuencia de forma implícita aun cuando no se utilicen explícitamente.

Implementación de pilas por listas enlazadas

La Figura 3.7 muestra el caso típico en el que basta con una pila pequeña, incluso aunque haya un gran número de operaciones. Si se está seguro de estar en este caso, entonces resulta apropiada la representación por array. Si no es así sólo una lista enlazada permitirá a la pila crecer y decrecer elegantemente, lo cual resulta especialmente útil si se trabaja con muchas estructuras de datos de este tipo. Para implementar operaciones básicas de pilas utilizando listas enlazadas, se comienza por definir la interfaz:

```
class Pila
{
public:
    Pila(int max);
    ~Pila();
    void meter(tipoElemento v);
    tipoElemento sacar();
    int vacia();
private:
    struct nodo
    { tipoElemento clave; struct nodo * siguiente; }
    struct nodo *cabeza, *z;
};
```

En C++, esta interfaz sirve para dos propósitos: para *utilizar* una pila, únicamente se necesita consultar la sección `public` de la interfaz para conocer qué operaciones se pueden realizar, y para *implementar* una rutina de pila se consulta la sección `private` para ver cuáles son las estructuras de datos básicas asociadas a la implementación. Las implementaciones de los procedimientos de pila se separan de la declaración de las clases, llegándose incluso a incluir en un archivo aparte. Esta capacidad para separar las implementaciones de las interfaces, y por lo tanto para experimentar fácilmente con diferentes implementaciones, es un aspecto muy importante de C++ que se tratará con mayor detalle al final de este capítulo.

Lo siguiente que se necesita es la función «constructor» para crear la pila cuando se declare y la función «destructor» para eliminarla cuando ya no se necesite (está fuera del alcance del libro):

```
Pila::Pila(int max)
{
    cabeza = new nodo; z = new nodo;
    cabeza->siguiente = z; z->siguiente = z;
```

```

    }
Pila::~Pila()
{
    struct nodo *t = cabeza;
    while (t != z)
        { cabeza = t; t->siguiente; delete cabeza; }
}

```

Para finalizar se muestra una implementación real de las operaciones de la pila:

```

void Pila::meter(tipoElemento v)
{
    struct nodo *t = new nodo;
    t->clave = v; t->siguiente = cabeza->siguiente;
    cabeza->siguiente = t;
}
tipoElemento Pila::sacar()
{
    tipoElemento x;
    struct nodo *t = cabeza->siguiente;
    cabeza->siguiente = t->siguiente; x = t->clave;
    delete t; return x;
}
int Pila::vacia()
{return cabeza->siguiente == z; }

```

Se aconseja al lector que estudie este código cuidadosamente para reforzar sus conocimientos, tanto de las listas enlazadas como de las pilas.

Colas

Otra estructura de datos de acceso restrictivo es la que se conoce como *cola*. En ella, una vez más, solamente se encuentran dos operaciones básicas: se puede insertar un elemento al principio de la cola y se puede eliminar un elemento del final. Quizás aquel ejecutivo tan ocupado podría organizar su trabajo como una cola, ya que entonces el trabajo que le llegue primero lo hará primero. En una pila algún elemento puede quedar sepultado en el fondo, pero en una cola todo se procesa en el orden en que se recibe.

La Figura 3.8 muestra un ejemplo de una cola que evoluciona mediante una sucesión de operaciones obtener y poner representadas por la sucesión

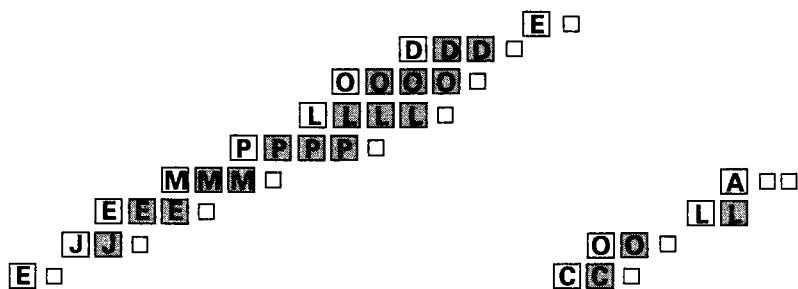


Figura 3.8 Características dinámicas de una cola.

*E * J E * M * P * LO * D*** E * C O *** LA ***.*

donde cada letra de la lista significa «poner» (la letra) y el asterisco significa «obtener».

Aunque en la práctica las pilas aparecen con mayor frecuencia que las colas, debido a su relación fundamental con la recursión (ver Capítulo 5), se encontrarán algoritmos para los que la cola es la estructura de datos natural. En el Capítulo 20 se encontrará una «cola de doble extremo» (deque), que es una combinación de pila y cola, y en los Capítulos 4 y 30 se verán ejemplos fundamentales que muestran cómo se puede utilizar una cola para realizar un mecanismo que permita examinar árboles y grafos. A veces se hará referencia a las pilas indicando que obedecen a la ley «last in, first out» (LIFO) (último en entrar, primero en salir) y, por su parte, las colas obedecen a la ley «first in, first out» (FIFO) (primero en entrar, primero en salir).

La implementación de las operaciones de colas por listas enlazadas es directa, y se deja como ejercicio para el lector. Al igual que ocurría con las pilas, también se puede utilizar un array si puede estimarse su tamaño máximo, como en la siguiente implementación de las funciones poner, obtener y vacía (se omite el código para la interfaz y las funciones constructor y destructor porque son similares al código dado anteriormente para la implementación de pilas por array):

```
void Cola::poner(tipoElemento v)
{
    cola[rabo++] = v;
    if (rabo > talla) rabo = 0;
}
tipoElemento Cola::obtener()
{
    tipoElemento t = cola[cabeza++];
}
```

```
    if (cabeza > talla) cabeza = 0;
    return t;
}
int Cola::vacia()
{ return cabeza == rabo; }
```

Hay tres variables de clase: la `talla` (tamaño) de la cola y dos índices, uno en `cabeza` y otro en «`rabo`» de la cola. El contenido de la cola está formado por todos los elementos del array entre `cabeza` y `rabo`, teniendo en cuenta la vuelta a 0 cuando se llegue al final del array. Si `cabeza` y `rabo` son iguales, entonces la cola se define como vacía; pero si poner las hiciera iguales, entonces se define como llena (aunque una vez más no se ha incluido esta comprobación en el código anterior). Esto requiere que el tamaño del array sea mayor, en una unidad, que el número máximo de elementos que se desea incluir en la cola: una cola llena contiene una posición del array vacía.

Tipos de datos abstractos y concretos

En lo anterior se ha visto que a menudo es conveniente describir los algoritmos y las estructuras de datos en función de las operaciones efectuadas, en lugar de hacerlo en términos de los detalles de la implementación. Cuando se define de esta forma una estructura de datos, se denomina *tipo de datos abstracto*. La idea es separar el «concepto» de lo que debe hacer la estructura de datos de cualquier implementación particular.

La característica genérica de un tipo de datos abstracto es que nada que sea externo a la definición de las estructuras de datos y los algoritmos que operan sobre ellas debe hacer referencia a cualquier cosa interna, excepto a través de llamadas a funciones y procedimientos de las operaciones fundamentales. El motivo principal para el desarrollo de los tipos de datos abstractos ha sido establecer un mecanismo para la organización de grandes programas. Los tipos de datos abstractos proporcionan una manera de limitar el tamaño y la complejidad de la interfaz entre algoritmos (potencialmente complicados), las estructuras de datos asociadas y los programas (un número potencialmente grande) que utilizan los algoritmos y las estructuras de datos. Esto hace más fácil comprender los grandes programas, y más conveniente los cambios o mejoras de los algoritmos fundamentales.

Las pilas y las colas son ejemplos clásicos de tipos de datos abstractos ya que muchos programas únicamente necesitan tratar con unas cuantas operaciones básicas bien definidas, y no con detalles de enlaces e índices.

Tanto los arrays como las listas enlazadas se pueden obtener por medio de mejoras de un tipo de datos abstracto básico denominado *lista lineal*. En cada uno de ellos se pueden realizar operaciones tales como *insertar*, *eliminar* y *ac-*

ceder en una estructura subyacente básica de elementos ordenados secuencialmente. Estas operaciones bastan para describir los algoritmos y la abstracción de listas lineales puede ser útil en las etapas iniciales de desarrollo del algoritmo. Pero, como se ha visto, el interés del programador reside en definir cuidadosamente qué operaciones se utilizarán, ya que puede haber bastantes características de rendimiento del algoritmo diferentes para implementaciones distintas. Por ejemplo, utilizar una lista enlazada en lugar de un array en la criba de Eratóstenes sería costoso porque la eficacia del algoritmo depende de que sea posible pasar rápidamente desde cualquier posición del array a cualquier otra, y utilizar un array en lugar de una lista enlazada en el problema de Josephus sería también costoso porque la eficacia del algoritmo depende de la desaparición de los elementos que se eliminan.

Las listas lineales sugieren otras muchas operaciones cuya eficacia requiere algoritmos y estructuras de datos mucho más sofisticados. Las dos operaciones más importantes son la *ordenación* de los elementos en orden creciente de sus claves (tema de los Capítulos 8-13), y la *búsqueda* de un elemento con una clave dada (tema de los Capítulos 14-18).

Un tipo de datos abstracto se puede utilizar para definir a otro. Así, se utilizaron las listas enlazadas y arrays para definir pilas y colas; de hecho se emplearon los conceptos de «puntero» y «registro» (proporcionados por C++) para construir listas enlazadas y el de «array» (proporcionado por C++) para construir arrays. Además, anteriormente se ha visto que se pueden construir listas enlazadas con arrays y en el Capítulo 36 se verá que ¡algunas veces los arrays se deben construir con listas enlazadas! El verdadero poder del concepto de tipo de datos abstracto es que permite construir sin inconvenientes grandes sistemas en diferentes niveles de abstracción: desde las instrucciones en lenguaje máquina proporcionadas por la computadora a las diversas posibilidades que proporciona el lenguaje de programación, o a la ordenación, la búsqueda y otras operaciones de mayor nivel proporcionadas por los algoritmos que se presentan en este libro hasta los niveles aún más altos de abstracción que pueda sugerir la aplicación.

En este libro se utilizarán programas relativamente pequeños que están muy relacionados con sus estructuras de datos asociadas. Siempre que sea posible se hablará en términos abstractos de la interfaz entre los algoritmos y sus estructuras de datos; es más apropiado enfocar el problema con un mayor nivel de abstracción (resulta más próximo a la aplicación): el concepto de abstracción no debe distraer de la búsqueda de la solución más eficaz de un problema concreto. ¡El rendimiento es lo que importa! Los programas que se han desarrollado teniendo esto en cuenta se pueden utilizar con cierta confianza al desarrollar los niveles de abstracción superiores de los grandes sistemas.

Las implementaciones de operaciones de colas y pilas de este capítulo son ejemplos de *tipos de datos concretos*, que guardan juntas las estructuras de datos y los algoritmos que operan sobre ellas. A lo largo de este libro se utilizará frecuentemente este paradigma, ya que es una forma muy conveniente de describir los algoritmos básicos, a la vez que desarrolla un código útil para emplearlo

en las aplicaciones. C++ proporciona un medio de implementar verdaderos tipos de datos abstractos utilizando la «jerarquía de clases» y «las funciones virtuales», en las que la interfaz consta solamente de funciones (*no* de la representación de los datos), pero en este libro no se utilizará este recurso porque la meta es el conocimiento de las características del rendimiento, lo cual es difícil de mantener cuando se utilizan verdaderos tipos de datos abstractos.

Como se mencionó anteriormente, las estructuras de datos reales rara vez constan simplemente de enteros y enlaces. Con frecuencia, los nodos contienen una gran cantidad de información y pueden pertenecer a múltiples estructuras de datos independientes. Por ejemplo, un archivo con los datos del personal, puede contener registros con nombres, direcciones y otros elementos de información sobre los empleados, y cada registro puede pertenecer a una estructura de datos destinada a la búsqueda de un empleado particular o a otra estructura de datos destinada al estudio de estadísticas, etc. C++ tiene un mecanismo general, denominado template, que proporciona una forma fácil de ampliar los algoritmos simples para trabajar en estructuras complejas. Si en lugar de utilizar `typedef` se coloca el código `template <class tipoElemento>` justo antes de las definiciones de clases de este capítulo, las convierte en definiciones de estructuras que trabajan con cualquier tipo de dato. Por ejemplo, esto permitiría una declaración como `Pila<float> acc()` utilizada para construir una pila de floats. En general, en este libro se trabajará con enteros, pero se mantendrán sin especificar los tipos, como en este capítulo, en el entendimiento de que `typedef` o `template` se pueden utilizar fácilmente en aquellas aplicaciones que se crea conveniente.

Ejercicios

1. Escribir un programa que llene un array bidimensional de valores booleanos poniendo $a[i][j]$ a 1 si el máximo común divisor de i y j es 1 y a 0 en cualquier otro caso.
2. Implementar una rutina `desplaza siguiente acabeza (struct nodo *t)` en una lista enlazada, para desplazar al comienzo de la lista el nodo siguiente al nodo que apunta `t`. (La Figura 3.3 es un ejemplo de esta operación para el caso especial en que `t` apunte al nodo siguiente al último de la lista.)
3. Implementar una rutina `intercambio (struct nodo *t, struct nodo *u)` en una lista enlazada, para intercambiar las posiciones de los nodos siguientes a los nodos apuntados por `t` y `u`.
4. Escribir un programa para resolver el problema de Josephus, utilizando un array en lugar de una lista enlazada.
5. Escribir procedimientos para insertar y eliminar en una lista doblemente enlazada.

6. Escribir procedimientos para la representación de una pila por una lista enlazada, pero utilizando arrays paralelos.
7. Obtener el contenido de la pila después de cada operación en la sucesión C U E * S * * T I O * * * N F * * * A * C I * L *. Aquí una letra significa «meter» (introducir la letra) y un «*» significa «sacar» (sacarla).
8. Obtener el contenido de la cola después de cada operación en la sucesión C U E * S * * T I O * * * N F * * * A * C I * L *. Aquí una letra significa «poner» (poner la letra) y un «*» significa «obtener» (tomarla).
9. Obtener una sucesión de llamadas a `eliminar` siguiente e `insertar` después que podría haber generado la Figura 3.5 desde una lista inicialmente vacía.
10. Implementar las operaciones básicas de una cola utilizando una lista enlazada.

Árboles

Las estructuras presentadas en el Capítulo 3 son intrínsecamente unidimensionales: un elemento sigue a otro. En este capítulo se considerarán las estructuras enlazadas de dos dimensiones denominadas *árboles*, que se encontrarán en el desarrollo de muchos de los algoritmos más importantes que se tratan a lo largo de este libro. Un estudio sobre árboles podría ocupar un libro entero, ya que se utilizan en muchas aplicaciones, además de en informática, y se han estudiado con profusión como objetos matemáticos. Desde luego, podría decirse que *este* libro es en sí mismo un tratado sobre árboles, ya que están presentes, de forma fundamental, en cada una de las secciones del libro. En este capítulo se estudiarán la terminología y las definiciones básicas asociadas a los árboles, se examinarán algunas propiedades importantes y se verá la forma de representar árboles mediante una computadora. En capítulos posteriores, se verán muchos algoritmos que operan con estas estructuras de datos elementales.

Los árboles se encuentran con frecuencia en la vida cotidiana, y el lector seguramente está familiarizado con el concepto básico. Por ejemplo, mucha gente hace el seguimiento de sus antepasados o descendientes, o ambas cosas, mediante un árbol genealógico, y así gran parte de la terminología se deriva de esta aplicación. Otro ejemplo es el caso de la organización de competiciones deportivas, que fue estudiado por Lewis Carroll y se verá en el Capítulo 11. Un tercer ejemplo es el organigrama de una gran empresa; este caso sugiere la «descomposición jerárquica» que aparece en muchas aplicaciones de la informática. Un cuarto ejemplo es el «árbol de análisis sintáctico» que descompone una oración gramatical en las partes que la forman; este proceso está íntimamente relacionado con el procesamiento de lenguajes de computadora, que se tratará en el Capítulo 21. A lo largo del libro se verán otros ejemplos.

Glosario

Este estudio de los árboles comienza definiéndolos como objetos abstractos e introduciendo la mayor parte de la terminología básica asociada. Los árboles se pueden definir de diferentes maneras, ya que hay una serie de propiedades matemáticas que implican esta equivalencia; esto se analizará con más detalle en la siguiente sección.

Un *árbol* es un conjunto no vacío de vértices y aristas que cumple una serie de requisitos. Un vértice es un objeto simple (también conocido como un *nodo*) que puede tener un nombre y puede llevar otra información asociada: una arista es una conexión entre dos vértices. Un *camino* en un árbol es una lista de vértices distintos en la que dos consecutivos se enlazan mediante aristas. A uno de los nodos del árbol se le designa como la *raíz*. La propiedad que define a un árbol es que hay exactamente un camino entre la raíz y cada uno de los otros nodos del árbol. Si hay más de un camino entre la raíz y un nodo, o si no existe ninguno entre la raíz y algún nodo, entonces se trata de un grafo (ver Capítulo 29) y no de un árbol. La Figura 4.1 muestra un ejemplo de árbol.

Aunque la definición no implica la «dirección» de la arista, normalmente se representan las aristas apuntando hacia afuera de la raíz (hacia abajo en la Figura 4.1) o hacia la raíz (hacia arriba en la Figura 4.1), dependiendo de la aplicación de la que se trate. Por lo regular, los árboles se dibujan con la raíz en la parte más alta (aunque en principio esto parezca antinatural), y se dice que el nodo y está debajo del nodo x (y x está por encima de y) si x está en el camino que va desde y a la raíz (es decir, y está debajo de x si existe un camino que lo conecte con x y que no pase por la raíz). Cada nodo (exceptuando la raíz) tiene exactamente un nodo inmediatamente encima de él, al que se denomina como su *padre*; mientras que a los nodos que tiene directamente por debajo se les denomina sus *hijos*. Algunas veces, siguiendo esta analogía familiar, se habla del «abuelo» o del «hermano» de un nodo: en la Figura 4.1, L es el nieto de E y tiene tres hermanos.

A los nodos sin hijos se les denomina a veces *hojas*, o *nodos terminales*. En correspondencia con su utilización posterior, a los nodos con al menos un hijo algunas veces se les denominará nodos *no terminales*. Los nodos terminales son

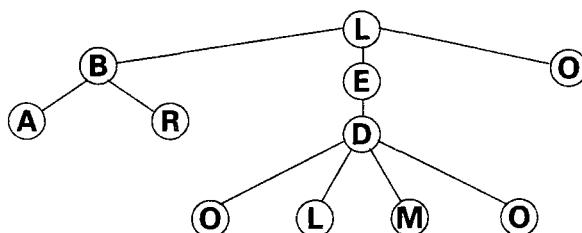


Figura 4.1 Un ejemplo de árbol.

frecuentemente diferentes de los no terminales: por ejemplo, pueden no tener nombre o información asociada. En tales situaciones, se hará referencia a los nodos no terminales como nodos *internos* y a los nodos terminales como nodos *externos*.

Cualquier nodo es la raíz de un *subárbol* constituido por él mismo y por los nodos situados debajo. En el árbol que se muestra en la Figura 4.1, hay siete subárboles de un nodo, un subárbol de tres nodos, un subárbol de cinco nodos y un subárbol de seis nodos. A un conjunto de árboles se le denomina *bosque*: por ejemplo, si se suprime la raíz y las aristas que la unen al árbol de la Figura 4.1, se obtiene un bosque formado por tres árboles de raíces B, E y O.

El orden en que se coloca a los hijos de un nodos es a veces significativo, y otras veces no. Un árbol *ordenado* es aquel en el que se ha especificado el orden de los hijos de todos los nodos. Por supuesto, los hijos se colocan en un orden determinado cuando se dibuja un árbol, y hay muchas formas diferentes de dibujar un árbol que no esté ordenado. Como se verá más adelante, es importante hacer esta distinción entre árboles ordenados y no ordenados a la hora de representar los árboles por computadora, ya que hay mucha menos flexibilidad en la representación de árboles ordenados. Naturalmente será la aplicación la que determine el tipo de árbol que se debe utilizar.

Los nodos de un árbol se estructuran en *niveles*: el nivel de un nodo es el número de nodos del camino que lleva desde éste hasta la raíz (sin incluirse a sí mismo). Así, por ejemplo, en la Figura 4.1, E es un nodo de nivel 1 y R es de nivel 2. La *altura* de un árbol es el nivel máximo del árbol (o la máxima distancia entre la raíz y cualquier nodo). La *longitud del camino* de un árbol es la suma de los niveles de todos los nodos del árbol (es decir, la suma de las longitudes de los caminos desde cada nodo a la raíz). El árbol de la Figura 4.1 tiene altura 3, y la longitud del camino es 21. Una vez que se han distinguido los nodos internos de los nodos externos, se puede hablar de la *longitud del camino interno* y de la *longitud del camino externo*.

Si cada nodo *debe* tener un número específico de hijos colocados en un orden determinado, entonces se tiene un *árbol multicamino*. En este tipo de árbol conviene definir nodos externos especiales que no tienen hijos (y normalmente ni nombre ni ninguna otra información asociada). En este caso los nodos externos actúan como nodos «ficticios» para referencia de nodos que no tienen el número de hijos especificado.

En particular, el caso más sencillo de árbol multicamino es el *árbol binario*, que es un árbol ordenado que está formado por dos tipos de nodos: nodos externos, sin hijos, y nodos internos, que tienen exactamente dos hijos. En la Figura 4.2 se muestra un ejemplo de un árbol binario. Como los dos hijos de cada nodo interno están ordenados, se hará referencia al *hijo de la izquierda* y al *hijo de la derecha*: todos los nodos internos deben tener los dos hijos, uno a la izquierda y otro a la derecha, aunque uno o los dos podrían ser nodos externos.

El objetivo de los árboles binarios es estructurar los nodos internos, ya que los externos sirven únicamente como «reserva de plaza» y se incluyen en la definición porque las representaciones más usuales de árboles binarios necesitan

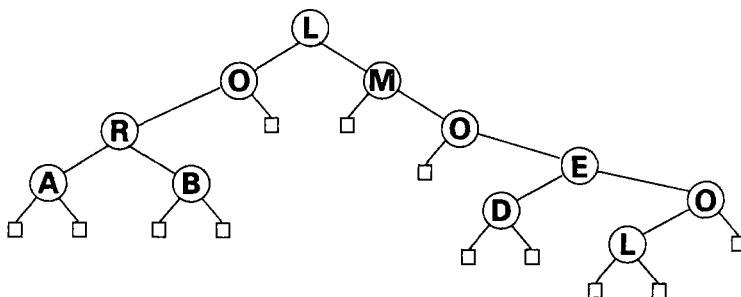


Figura 4.2 Un ejemplo de árbol binario.

conocer a todos sus nodos externos. Un árbol binario puede estar «vacío» si contiene un nodo externo o ninguno interno.

Un árbol binario *lleno* es aquel en el que los nodos internos llenan todos los niveles, con la posible excepción del último. Un árbol binario *completo* es un árbol binario lleno en el que los nodos internos del último nivel aparecen todos a la izquierda de los nodos externos de ese mismo nivel. La Figura 4.3 muestra un ejemplo de un árbol binario completo. Como se verá más adelante, los árboles binarios aparecen muchas veces en aplicaciones de computadoras, siendo mejor su rendimiento cuando están llenos (o casi llenos). En el Capítulo 11 se estudiará una estructura de datos basada en los árboles binarios completos.

El lector debería observar con gran cuidado que, mientras que todo árbol binario es un árbol, no todo árbol es un árbol binario. Incluso considerando únicamente árboles ordenados en los que todos los nodos tienen 0, 1 o 2 hijos, cada uno de ellos puede corresponder a muchos árboles binarios, porque los nodos con 1 hijo pueden estar a la izquierda o a la derecha de un árbol binario.

En el próximo capítulo se verá que los árboles están íntimamente ligados con la recursión. De hecho, la forma más sencilla de definir árboles es la recursiva, como se indica a continuación: «un árbol es o un nodo aislado o un nodo raíz conectado a un conjunto de árboles» y «un árbol binario es o un nodo externo o un nodo raíz (interno) conectado a un árbol binario izquierdo y a un árbol binario derecho».

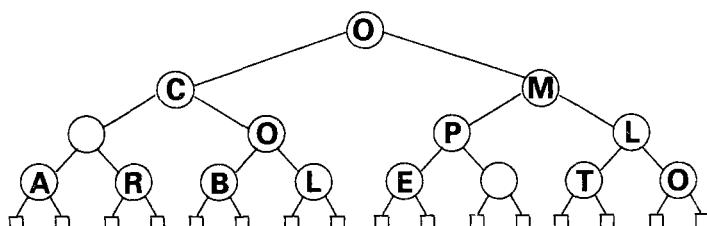


Figura 4.3 Un árbol binario completo.

Propiedades

Antes de tratar las representaciones es preciso ver el aspecto matemático, considerando una serie de propiedades importantes de los árboles. Una vez más aparece un gran número de propiedades a examinar, pero aquí con el objeto de tratar aquellas que son particularmente importantes para los algoritmos que se verán en el libro.

Propiedad 4.1 *Dados dos nodos cualesquiera de un árbol, existe exactamente un camino que los conecta.*

Dados dos nodos cualesquiera, se verifica que tienen un *antecesor común mínimo*, es decir un nodo que está en el camino de ambos nodos hacia la raíz pero de manera que ninguno de sus hijos tiene esta misma propiedad. Por ejemplo, C es el antecesor común más pequeño de R y L en el árbol de la Figura 4.3. El antecesor común mínimo debe existir siempre porque, o bien es la raíz, o bien ambos nodos están en un subárbol enraizado en uno de los hijos de la raíz; en este último caso, o bien ese nodo es el antecesor común mínimo, o ambos nodos están en el subárbol enraizado en uno de sus hijos, etc. Existe un camino desde cada uno de los nodos al antecesor común mínimo: componiendo estos dos caminos se obtiene un camino que conecta a los dos nodos. ■

Una consecuencia importante de la propiedad 4.1 es que cualquier nodo puede ser la raíz, ya que, dado cualquier nodo de un árbol, existe exactamente un camino que lo conecta con cualquier otro nodo del árbol. Técnicamente la definición en la que la raíz está identificada es la del *árbol enraizado u orientado*. A un árbol en el que la raíz no está identificada se le denomina *árbol libre*. El lector no necesita saber más sobre estas diferencias: o la raíz está identificada, o no lo está.

Propiedad 4.2 *Un árbol con N nodos tiene $N - 1$ aristas.*

Esta propiedad se deduce directamente del hecho de que cada nodo, excepto la raíz, tiene un único parente, y toda arista conecta a un nodo con su parente. También es posible probar este hecho por inducción a partir de la definición recursiva. ■

Las dos siguientes propiedades pertenecen a los árboles binarios. Como se mencionó anteriormente, estas estructuras se encontrarán muy a menudo a lo largo del libro, de manera que merece la pena prestar atención a sus características. Esto servirá como trabajo preparatorio para comprender el comportamiento representativo de diversos algoritmos que se encontrarán más adelante.

Propiedad 4.3 *Un árbol binario con N nodos internos tiene $N + 1$ nodos externos.*

Esta propiedad se puede demostrar por inducción. Un árbol binario sin nodos internos tiene un nodo externo, de manera que la propiedad se verifica para $N = 0$. Para $N > 0$, cualquier árbol binario con N nodos internos tienen k nodos

internos en el subárbol de la izquierda y $N - 1 - k$ nodos internos en el subárbol de la derecha para k entre 0 y $N - 1$, ya que la raíz es un nodo interno. Por la hipótesis de inducción, el subárbol izquierdo tiene $k + 1$ nodos externos y el subárbol derecho tiene $N - k$ nodos externos, lo que hace un total de $N + 1$. ■

Propiedad 4.4 *La longitud del camino externo de cualquier árbol binario con N nodos internos es $2N$ mayor que la longitud del camino interno.*

Esta propiedad también se puede probar por inducción, pero es igualmente instructiva una demostración alternativa. Se observa que cualquier árbol binario puede construirse mediante el siguiente proceso: se comienza con un árbol binario formado por un nodo externo, a continuación se repite la siguiente operación N veces: coger un nodo externo y reemplazarlo por un nuevo nodo interno con dos nodos externos como hijos; si el nodo externo elegido es de nivel k , la longitud del camino interno aumenta en k , pero la longitud del camino externo aumenta en $k+2$ (se ha eliminado un nodo externo de nivel k , pero se han añadido dos de nivel $k + 1$). El proceso comienza con un árbol cuya longitud de camino externo e interno es 0 y que en cada uno de los N pasos aumenta la longitud del camino externo 2 unidades más que la del camino interno. ■

Finalmente, se considera una propiedad elemental de la «mejor» clase de árboles binarios, los árboles llenos. Estos árboles son interesantes porque garantizan que su altura será baja, de manera que no costará mucho trabajo ir de la raíz a cualquier nodo o viceversa.

Propiedad 4.5 *La altura de un árbol binario lleno con N nodos internos es aproximadamente $\log_2 N$.*

Haciendo referencia a la Figura 4.3, si la altura es n se debe tener

$$2^{n-1} < N + 1 \leqslant 2^n,$$

ya que hay $N + 1$ nodos externos. Esto implica la propiedad enunciada. (En realidad la altura es exactamente igual al resultado de redondear $\log_2 N$ al entero más próximo, pero, como se verá en el Capítulo 6, no hay que ser tan preciso.) ■

Otras propiedades matemáticas de los árboles se irán presentando según se vayan necesitando en los capítulos posteriores. En este momento ya se puede comenzar con las consideraciones prácticas de la representación de los árboles en la computadora y su manipulación eficaz.

Representación de árboles binarios

La representación más frecuente de los árboles binarios consiste en utilizar registros con *dos* enlaces por nodo. Normalmente, se llamará a los enlaces izq y der (izquierda y derecha) para indicar el orden elegido en la representación que corresponde a la forma en que se ha dibujado el árbol en la página. En algunas

aplicaciones puede resultar apropiado tener dos tipos de registros diferentes, uno para los nodos internos y otro para los externos; en otras puede ser más adecuado utilizar sólo un tipo de nodo y emplear los enlaces a los nodos externos para otros propósitos.

Como modelo de la construcción y utilización de los árboles binarios se continuará con el ejemplo del capítulo anterior, que trata del procesamiento de expresiones aritméticas. Como se muestra en la Figura 4.4, hay una correspondencia directa entre las expresiones aritméticas y los árboles.

Se emplearán como identificadores de los argumentos caracteres sencillos en lugar de números (la razón se explicará más adelante). El *árbol de análisis sintáctico* de una expresión se define por la siguiente regla recursiva: «poner el operador en la raíz y a continuación construir el árbol de la expresión correspondiente al primer operando a la izquierda y el de la expresión correspondiente al segundo operando a la derecha. La Figura 4.4 es el árbol de análisis sintáctico de $A \cdot B C + D E ^\star F + ^\star$ (la misma expresión en postfija). Obsérvese que infija y postfija son dos formas de representar expresiones aritméticas, siendo los árboles de análisis sintáctico una tercera.

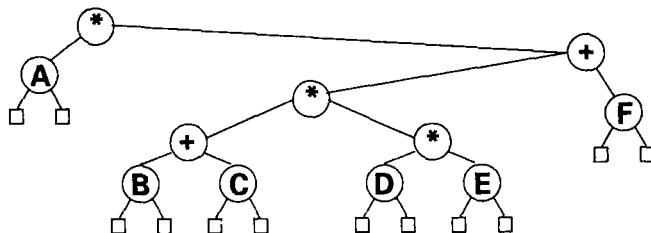


Figura 4.4 Árbol de análisis sintáctico para $A \cdot ((B + C) \cdot (D \cdot E)) + F$.

Como los operadores tienen exactamente dos operandos, lo adecuado para este tipo de expresiones es un árbol binario, pero otras expresiones más complicadas podrían necesitar un tipo de árbol diferente. En el Capítulo 21 se revisarán estos resultados con más detalle, aunque por ahora el objetivo es la simple construcción de un árbol que represente una expresión aritmética.

El siguiente código construye el árbol de análisis sintáctico de una expresión aritmética a partir de una representación de entrada postfija. Se trata de una ligera modificación del programa del capítulo anterior que evaluaba expresiones postfijas utilizando una pila. En lugar de guardar los resultados de los cálculos intermedios en la pila, se guarda la expresión de los árboles, como en la siguiente implementación:

```

struct nodo
{
    char info;
    struct nodo *izq, *der;
};
struct nodo *x, *z;
char c; Pila pila(50);
  
```

```

z = new nodo; z->izq = z; z->der = z;
while (cin.get(c))
{
    while (c == ' ') cin.get(c);
    x = new nodo;
    x->info = c; x->izq = z; x->der = z;
    if (c=='+' || c=='*')
        { x->der = pila.sacar(); x->izq = pila.sacar(); }
    pila.meter(x);
}

```

Se utiliza el tipo de pila del Capítulo 3, con un declaración `typedef` apropiada de modo que se ponen en la pila punteros en lugar de enteros. Todos los nodos tienen un carácter y dos enlaces a otros nodos. Cada vez que se encuentra un nuevo carácter distinto del espacio en blanco, se crea un nodo utilizando el operador estándar de C++ `new` que llama a un constructor y asigna espacio en la memoria. Si se trata de un operador, los subárboles de sus operandos están en lo más alto de la pila, como en una evaluación postfija. Si es un operando, entonces sus enlaces son nulos. En vez de utilizar enlaces nulos, se utilizará un nodo ficticio `z` cuyos enlaces apuntan a él mismo. Esto facilita la representación de ciertas operaciones mediante árboles (ver Capítulo 14). La Figura 4.5 muestra los pasos intermedios de la construcción del árbol de la Figura 4.4.

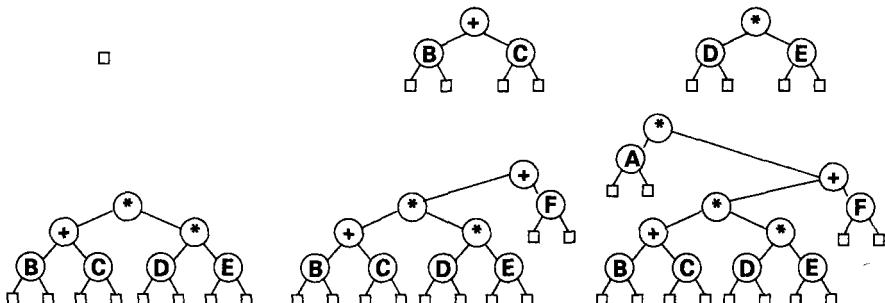


Figura 4.5 Construcción del árbol de análisis sintáctico de $A B C + D E * * F + *$.

Este programa, más bien sencillo, puede modificarse para tratar expresiones más complicadas que utilicen operadores con un único argumento, tales como la exponentiación. El procedimiento es muy general; es exactamente el mismo que se utiliza, por ejemplo, para analizar y compilar programas en C++. Una vez creado el árbol de análisis sintáctico se puede utilizar para muchas cuestiones, como para evaluar la expresión o crear programas de computadora para evaluarla. En el Capítulo 21 se presentarán procedimientos generales para construir árboles de análisis; en éste se verá cómo se puede utilizar un árbol para

evaluar la expresión, si bien el interés de este capítulo se centra más en el procedimiento de construcción del árbol.

Al igual que en las listas enlazadas, siempre existe la alternativa de utilizar arrays paralelos en lugar de punteros y registros para implementar la estructura de datos de un árbol binario. Como se dijo con anterioridad, esto es especialmente útil cuando se conoce de antemano el número de nodos; incluso en el caso particular en que los nodos necesiten ocupar un array para algún otro propósito se acudirá a esta alternativa.

La representación por doble enlace de los árboles binarios antes utilizados permite *descender* por el árbol, pero no proporciona una forma de *ascender* por él. La situación es análoga a la de las listas enlazadas simples frente a las listas doblemente enlazadas: se puede añadir otro enlace a cada nodo para permitir más libertad de movimientos, pero con el coste de una implementación más complicada. Existen otras opciones diferentes entre las estructuras de datos avanzadas que facilitan el movimiento a lo largo del árbol; pero para los algoritmos de este libro generalmente bastará la representación de doble enlace.

En el programa anterior se utilizó un nodo «ficticio» en lugar de nodos externos. Al igual que en las listas enlazadas, este cambio será conveniente en muchas situaciones, pero no siempre será apropiado, pues hay otras dos soluciones comúnmente utilizadas. Una de ellas consiste en emplear un tipo diferente de nodo sin enlaces, para los nodos externos. Otra solución consiste en marcar los enlaces de alguna manera (para distinguirlos de otros enlaces del árbol), y hacer que apunten a otra parte del árbol (un método de este tipo se expondrá a continuación). Este tema se revisará en los Capítulos 14 y 17.

Representación de bosques

Los árboles binarios tienen dos enlaces debajo de cada nodo interno, de manera que la representación de árboles empleada anteriormente para ellos es inmediata. Pero ¿qué hacer con los árboles en general, o con los bosques, en los que cada nodo puede tener un número aleatorio de enlaces hacia su descendencia? La respuesta es que existen dos maneras relativamente sencillas de salir de este dilema.

En primer lugar, en muchas aplicaciones no se necesita recorrer el árbol hacia abajo, sino *únicamente hacia arriba!* En tales casos, sólo se necesita un enlace para cada nodo; el que lo conecta a su padre. La Figura 4.6 muestra esta representación para el árbol de la Figura 4.1: el array *a* contiene la información asociada a cada registro y el array *papa* contiene los enlaces padre. Así, la información asociada al padre de *a[i]* está en *a[papa[i]]*, etc. Por convenio se hace que la raíz se apunte a sí misma. Ésta es una representación bastante compacta y muy recomendable para trabajar con árboles si sólo se recorren hacia arriba. Se verán ejemplos de la utilización de esta representación en los Capítulos 22 y 30.

k	1	2	3	4	5	6	7	8	9	10	11
a [k]	(A)	(R)	(B)	(O)	(L)	(M)	(O)	(D)	(E)	(L)	(O)
papa [k]	3	3	10	8	8	8	8	9	10	10	10

Figura 4.6 Representación del enlace padre de un árbol.

Para representar bosques por el procedimiento descendente se necesita una forma de tratar los hijos de cada nodo sin tener que asignar de antemano un número determinado de hijos para cada uno. Pero éste es exactamente el tipo de restricción para la que se diseñaron las listas enlazadas. Claro está, debe utilizarse una lista enlazada para los hijos de cada nodo. Entonces cada nodo contiene dos enlaces, uno para la lista enlazada que lo conecta con sus hermanos y otro para la lista enlazada de sus hijos. La Figura 4.7 muestra esta representación para el árbol de la Figura 4.1. En lugar de utilizar un nodo ficticio para terminar cada lista, es preferible obligar a que el último nodo vuelva a apuntar hacia su padre; de esta manera es posible moverse a través del árbol, tanto hacia arriba como hacia abajo. (Estos enlaces pueden marcarse para distinguirlos de los enlaces «hermanos»; de forma alternativa, se pueden explorar completamente los hijos de un nodo marcando o guardando el nombre del padre de manera que se pueda interrumpir la exploración cuando se vuelva a encontrar al padre.)

Pero en esta representación cada nodo tiene exactamente dos enlaces (uno hacia el hermano de la derecha y otro hacia el hijo que está más a la izquierda). Se puede preguntar si existe alguna diferencia entre esta estructura de datos y un árbol binario. La respuesta es que no existe, como se muestra en la Figura 4.8 (el árbol de la Figura 4.1 representado como un árbol binario). Esto es, cualquier bosque puede representarse por un árbol binario haciendo que el enlace de la izquierda de cada nodo apunte al hijo que queda más a la izquierda y que el enlace de la derecha de cada nodo apunte a su hermano de la derecha. (Este hecho sorprende muy a menudo al principiante.)

Por tanto, es posible utilizar bosques siempre que sea conveniente para el diseño del algoritmo. Cuando se quiere ascender por el árbol, la representación

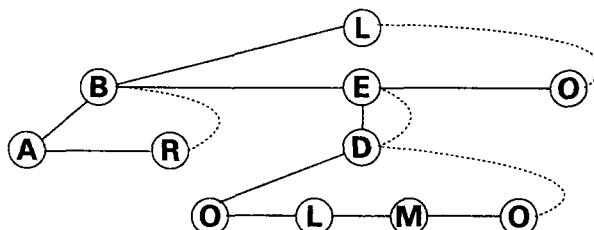


Figura 4.7 Representación de un árbol por el hijo más a la izquierda y el hermano de la derecha.

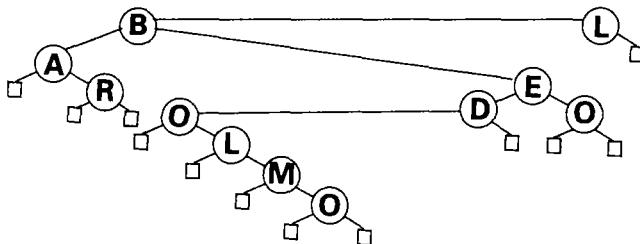


Figura 4.8 Representación de un árbol mediante un árbol binario.

del enlace padre hace que los bosques sean más fáciles de tratar que cualquier otra clase de árbol, y cuando se desea descender los bosques son esencialmente equivalentes a los árboles binarios.

Recorrido de los árboles

Una vez que se ha construido el árbol, lo primero que se necesita es saber cómo *recorrerlo*, es decir, cómo visitar sistemáticamente todos los nodos. Esta operación es trivial para las listas lineales por su definición, pero para los árboles existen diferentes formas de hacerlo que difieren sobre todo en el *orden* en que se recorren los nodos. Como se verá, el orden más apropiado depende de cada aplicación concreta.

Por el momento se estudia el recorrido de árboles binarios; debido a la equivalencia entre bosques y árboles binarios, los métodos descritos son también útiles para bosques, aunque más adelante también se mencionarán métodos que se aplican directamente a los bosques.

El primer método a considerar es el recorrido en *orden previo* (*preorden*), que puede utilizarse, por ejemplo, para escribir la expresión representada por el árbol de la Figura 4.4 en prefijo. Este método se define mediante una simple regla recursiva: «visitar la raíz, después el subárbol izquierdo y a continuación el subárbol derecho.» Se mostrará en el próximo capítulo la implementación más sencilla de este método, la recursiva, que está estrechamente relacionada con la siguiente implementación basada en una pila:

```

recorrer(struct nodo *t)
{
    pila.meter(t);
    while (!pila.vacia())
    {
        t = pila.sacar(); visitar(t);
    }
}

```

```

        if (t->der != z) pila.meter(t->der);
        if (t->izq != z) pila.meter(t->izq);
    }
}

```

Siguiendo la regla, «se visita un subárbol» después de visitar primero la raíz. Como no se pueden visitar los dos subárboles a la vez, se guarda el subárbol derecho en una pila y se visita el subárbol izquierdo. Cuando termine esta visita el subárbol derecho estará en lo más alto de la pila y podrá visitarse. La Figura 4.9 muestra este programa aplicado al árbol binario de la Figura 4.2: el orden en el que se visitan los nodos es L O R A B M O E D O L.

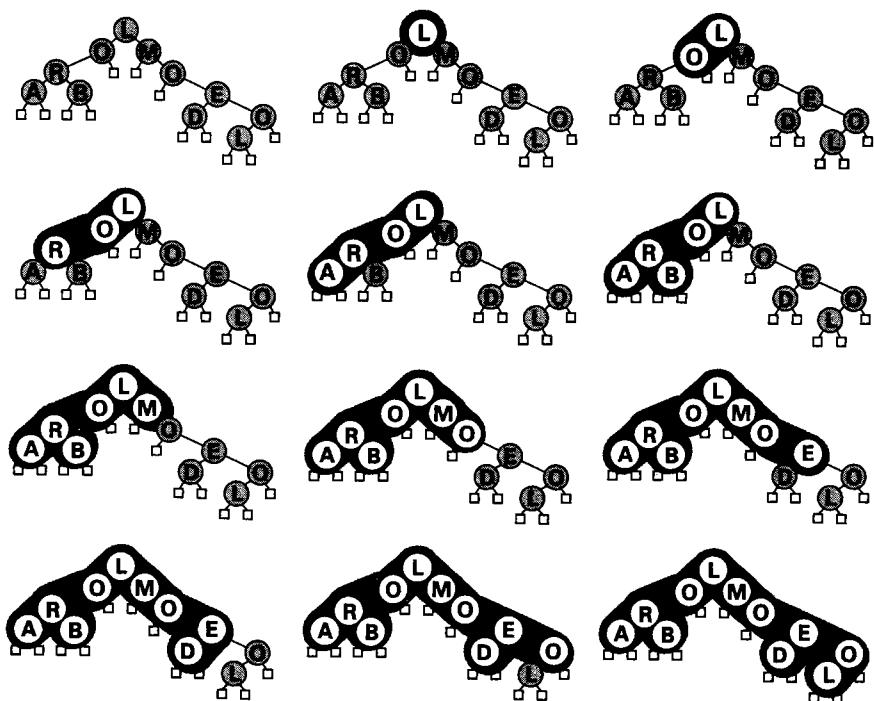


Figura 4.9. Recorrido en orden previo (preorden).

Para demostrar que efectivamente el programa visita los nodos del árbol por orden previo, se puede emplear el método de inducción, tomando como hipótesis inductiva que los subárboles se visitan en orden previo y que el contenido de la pila justo antes de visitar un subárbol es el mismo que justo después de visitarlo.

El segundo método a considerar es el recorrido *en orden*, que puede utilizarse, por ejemplo, para escribir las expresiones aritméticas en infija correspondientes a los árboles de análisis sintáctico (con algún trabajo extra para obtener

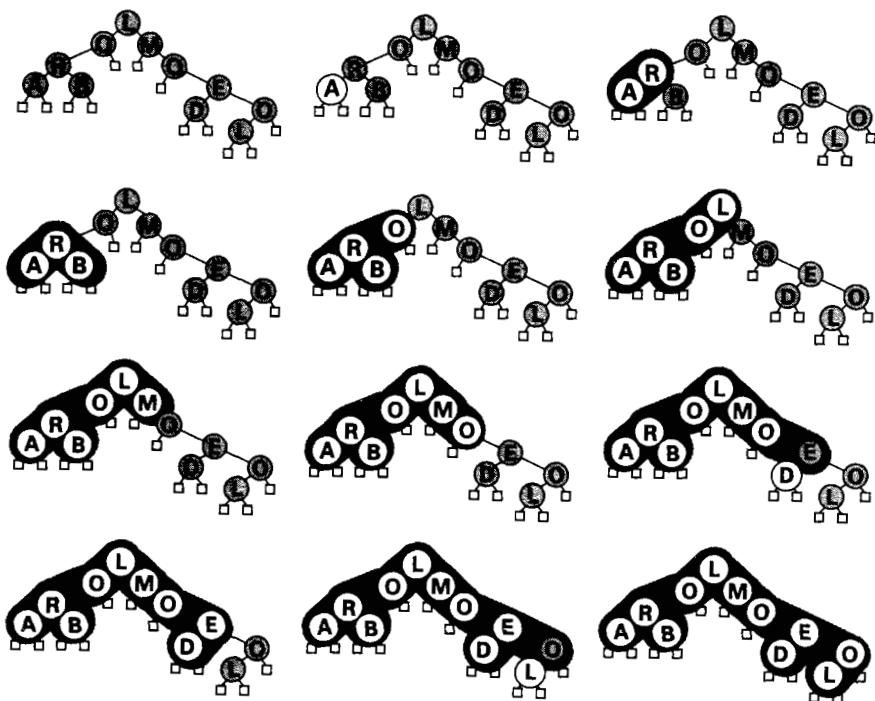


Figura 4.10. Recorrido en orden simétrico.

los paréntesis de la derecha). Al igual que en el orden previo, el recorrido en orden se define con la regla recursiva «visitar el subárbol izquierdo, a continuación la raíz y después el subárbol derecho». A veces a este método también se le denomina recorrido en *orden simétrico*, por razones evidentes. La implementación de un programa (basado en una pila) para recorrer el árbol por orden simétrico es casi idéntica al programa anterior; se omite aquí porque constituye el tema principal del siguiente capítulo. En la Figura 4.10 se ve cómo se visitan en orden simétrico los nodos del árbol de la Figura 4.2: los nodos se visitan en el orden A R B O L M O D E L O. Este método de recorrido es probablemente el más utilizado, ya que, por ejemplo, ejerce un papel fundamental en las aplicaciones de los Capítulos 14 y 15.

El tercer tipo de recorrido recursivo, llamado *orden posterior (postorden)*, se define mediante la siguiente regla recursiva: «visitar el subárbol izquierdo, a continuación el subárbol derecho y después la raíz». La Figura 4.11 muestra cómo se visitan los nodos del árbol de la Figura 4.2 en orden posterior: A B R O D L O E O M L. Si se visita el árbol de la Figura 4.4 en orden posterior se obtiene la expresión $A B C + D E * * F + *$, como era de esperar. La implementación de un programa (basado en una pila) para recorrer un árbol en orden posterior es más complicada que la de los otros dos recorridos, porque el pro-

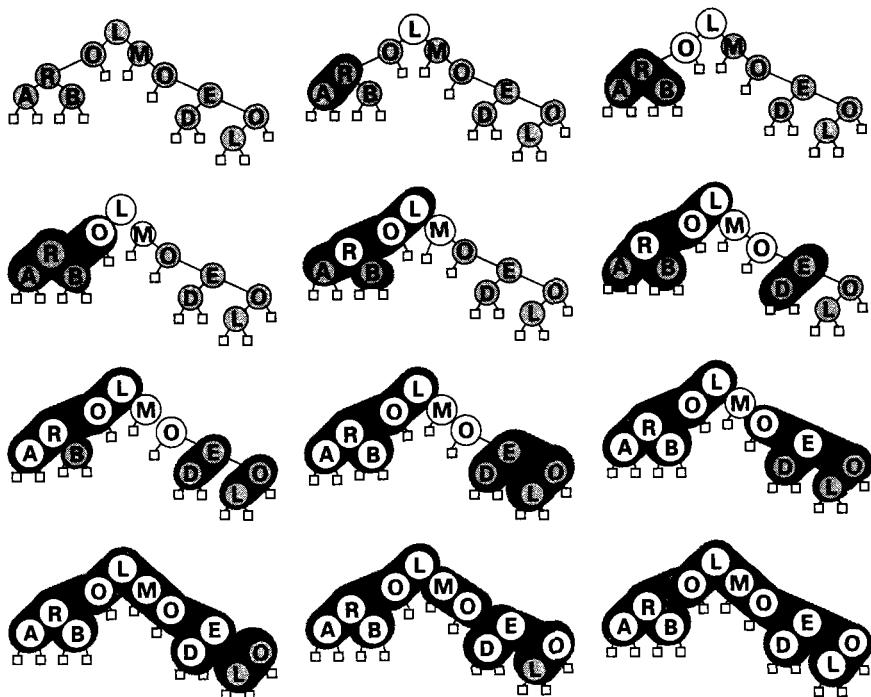


Figura 4.11. Recorrido en orden posterior (*postorden*).

grama debe organizarse para guardar la raíz y el subárbol derecho mientras se visita el subárbol izquierdo, y para guardar la raíz mientras se visita el subárbol derecho. Los detalles de esta implementación se dejan como ejercicio para el lector.

La cuarta estrategia de recorrido que se considera no es del todo recursiva, ya que simplemente se visitan los nodos según van apareciendo en la página, leyendo de arriba abajo y de izquierda a derecha. Este método se denomina recorrido en *orden de nivel* porque todos los nodos de cada nivel aparecen juntos, en orden. La Figura 4.12 muestra cómo se visitan los nodos del árbol de la Figura 4.2 si se recorren por orden de nivel.

Singularmente, el recorrido en orden de nivel se puede conseguir empleando el programa anterior para orden previo, utilizando una cola en lugar de una pila:

```
recorrer(struct nodo *t)
{
    cola.poner(t);
    while (!cola.vacia())
    {
        t = cola.obtener(); visitar(t);
```

```

    if (t->izq != z) cola.poner(t->izq);
    if (t->der != z) cola.poner(t->der);
}
}

```

Por una parte este programa es virtualmente idéntico al anterior, ya que la única diferencia es que éste utiliza una estructura de datos FIFO mientras que el otro utiliza una estructura de datos LIFO. Por la otra, estos programas procesan los árboles de forma fundamentalmente diferente. Estos programas merecen un estudio cuidadoso, ya que muestran las diferencias esenciales entre las pilas y las colas. Se volverá sobre este tema en el Capítulo 30.

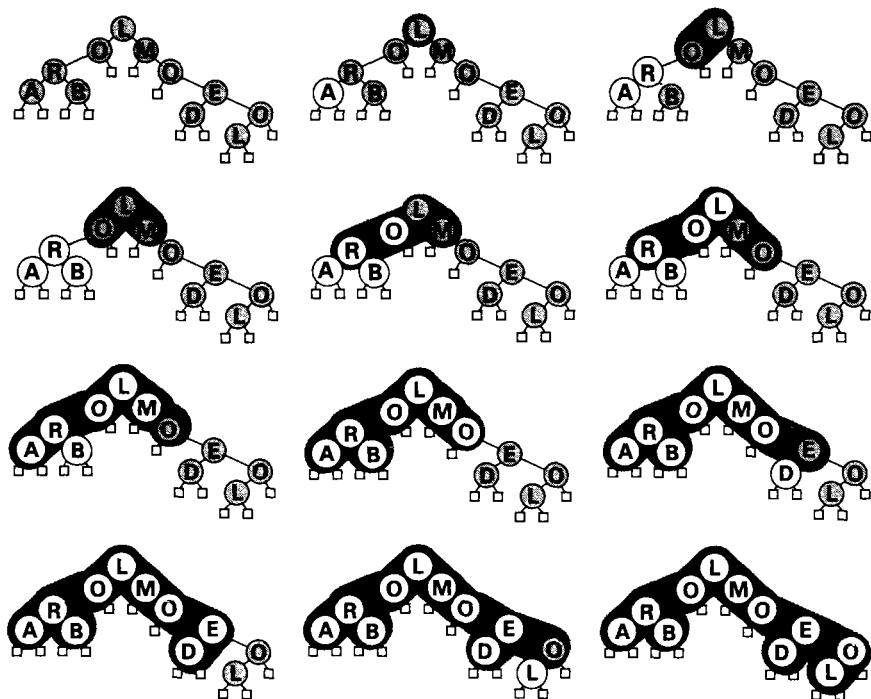


Figura 4.12. Recorrido en orden de nivel.

Los recorridos en orden previo, orden posterior y orden de nivel también se pueden definir para bosques. Para hacer coherentes las definiciones, basta con representar un bosque como un árbol con una raíz imaginaria. Entonces la regla del orden previo queda como «visitar la raíz y después cada uno de los subárboles»; y la regla para el recorrido en orden posterior queda como «visitar cada uno de los subárboles y después la raíz». La regla del recorrido por orden de nivel es la misma que para los árboles binarios. Cabe destacar el hecho de que el orden previo para un bosque es el mismo que el orden previo para su corres-

pondiente árbol binario, como se vio anteriormente, y que el orden posterior para un bosque es igual que el recorrido por orden simétrico para el árbol binario; pero no ocurre lo mismo para el recorrido por orden de nivel. Las implementaciones directas que utilizan pilas y colas son sencillas generalizaciones de los programas dados con anterioridad para los árboles binarios.

Ejercicios

1. Indicar el orden en el que se visitarán los nodos del árbol de la Figura 4.3 si se recorre en orden previo, orden simétrico, orden posterior y orden de nivel.
2. ¿Cuál es la altura de un árbol cuaternario completo con N nodos?
3. Dibujar el árbol de análisis sintáctico de la expresión $(A + B) * C + (D + E)$.
4. Considerando el árbol de la Figura 4.2 como un bosque que se ha representado como un árbol binario; dibujar esa representación.
5. Indicar el contenido de la pila cada vez que se visita un nodo durante el recorrido en orden previo representado en la Figura 4.9.
6. Indicar el contenido de la cola cada vez que se visita un nodo durante el recorrido en orden de nivel representado en la Figura 4.12.
7. Dar un ejemplo de un árbol para el que la pila utiliza más espacio al recorrerlo en orden previo que la cola al recorrerlo en orden de nivel.
8. Dar un ejemplo de un árbol para el que la pila utiliza menos espacio al recorrerlo en orden previo que la cola al recorrerlo en orden de nivel.
9. Dar una implementación basada en una pila del recorrido en orden posterior de un árbol binario.
10. Escribir un programa para implementar un recorrido en orden de nivel de un bosque representado como un árbol binario.

Recursión

La recursión es un concepto fundamental en matemáticas e informática. La definición más sencilla es que un programa recursivo es aquel que se llama a sí mismo (y una función recursiva es aquella que se define en términos de sí misma). Sin embargo, como un programa recursivo no puede llamarse a sí mismo siempre, porque nunca se detendría (y una función recursiva no puede definirse siempre en términos de sí misma, o la definición sería cíclica), otro ingrediente esencial de la definición es que debe contener una *condición de terminación* que autoriza al programa a dejar de llamarse a sí mismo (y a que la función deje de definirse en términos de sí misma). Todos los cálculos prácticos pueden expresarse de una forma recursiva.

El primer objetivo de este capítulo será examinar la recursión como una herramienta práctica. En primer lugar, se darán algunos ejemplos en los que la recursión *no* es práctica, mientras se muestra la relación entre recurrencias matemáticas simples y programas recursivos simples. Después, se mostrará un ejemplo prototípico de un programa recursivo de «divide y vencerás» del tipo que se utiliza para resolver problemas fundamentales en secciones posteriores de este libro. Finalmente, se estudiará cómo puede eliminarse la recursión de cualquier programa recursivo, y se mostrará un ejemplo detallado de cómo eliminar la recursión de un algoritmo de recorrido de árbol recursivo simple para obtener un algoritmo no recursivo simple basado en una pila.

Como se verá más adelante, muchos algoritmos interesantes se pueden expresar fácilmente mediante programas recursivos y muchos diseñadores de algoritmos prefieren métodos recursivos. Pero también es muy frecuente el caso de que un algoritmo tan interesante como los anteriores se encuentre escondido en los detalles de una implementación (necesariamente) no recursiva —en este capítulo se estudiarán las técnicas que permiten encontrar tales algoritmos—.

Recurrencias

Las definiciones recursivas de funciones son muy frecuentes en matemáticas; el tipo más simple, en el que intervienen argumentos enteros, se denomina *relaciones de recurrencia*. Quizás la función más familiar de este tipo sea la función *factorial*, definida por la fórmula

$$N! = N \cdot (N - 1)!, \quad \text{para } N \geq 1 \text{ con } 0! = 1.$$

Esta definición se corresponde directamente con el siguiente programa recursivo simple:

```
int factorial(int N)
{
    if (N == 0) return 1;
    return N * factorial(N-1);
}
```

Por una parte, este programa ilustra los aspectos básicos de un programa recursivo: se llama a sí mismo (con un valor más pequeño que su argumento) y tiene una condición de terminación en la que calcula directamente su resultado. Por otra parte, no se oculta el hecho de que este programa es tan sólo un bucle *for* con adornos, por lo que difícilmente puede ser un ejemplo convincente del poder de la recursión. También es importante recordar que es un *programa* y no una ecuación: por ejemplo, ni la ecuación ni el programa anterior «funcionan» para un *N* negativo, pero los efectos negativos de esta omisión son quizá más perceptibles con el programa que con la ecuación. La llamada a *factorial(-1)* se traduce en un bucle infinito recursivo; éste es, de hecho, un fallo recurrente que puede aparecer de forma más sutil en programas recursivos más complejos.

Una segunda relación de recurrencia muy conocida es la que define los *números de Fibonacci*:

$$F_N = F_{N-1} + F_{N-2}, \quad \text{para } N \geq 2 \text{ con } F_0 = F_1 = 1.$$

Esto define la sucesión

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \dots$$

De nuevo, la recurrencia se corresponde directamente con un programa recursivo simple:

```
int fibonacci(int N)
```

```
{
    if (N <= 1) return 1;
    return fibonacci(N-1) + fibonacci(N-2);
}
```

Éste es un ejemplo todavía menos convincente de la «potencia» de la recursión; en efecto, es un ejemplo convincente de que la recursión no debería utilizarse a ciegas, ya que puede resultar dramáticamente ineficaz. El problema aquí es que las llamadas recursivas indican que F_{N-1} y F_{N-2} deben calcularse independientemente, cuando, de hecho, lo natural sería utilizar F_{N-2} (y F_{N-3}) para calcular F_{N-1} . En realidad, es fácil calcular cuál es el número exacto de llamadas al procedimiento `fibonacci` anterior que se necesitan para calcular F_N : el número de llamadas necesarias para calcular F_N es el número de llamadas necesarias para calcular F_{N-1} más el número de llamadas necesarias para calcular F_{N-2} , a menos que $N = 0$ o $N = 1$, en cuyo caso sólo se necesita una llamada. Pero esto encaja exactamente con la relación de recurrencia que define los números de Fibonacci: el número de llamadas a `fibonacci` para calcular F_N . Se sabe que F_N es aproximadamente ϕ^N , donde $\phi = 1,61803\dots$ es la «razón de oro»: la tremenda verdad es que ¡el programa anterior es un algoritmo de *tiempo exponencial* para calcular los números de Fibonacci!

Por el contrario, es muy fácil calcular F_N en tiempo lineal, como se indica a continuación:

```
const int max = 25;
int fibonacci (int N)
{
    int i, F[max];
    F[0] = 1; F[1] = 1;
    for (i = 2; i <= max; i++)
        F[i] = F[i-1] + F[i-2];
    return F[N];
}
```

Este programa calcula los primeros `max` números de Fibonacci, utilizando un array de tamaño `max`. (Como los números crecen exponencialmente, `max` será pequeño.)

De hecho, esta técnica de utilizar un array para almacenar resultados previos es el método que suele elegirse para evaluar relaciones de recurrencia, ya que permite resolver de manera eficaz y uniforme las ecuaciones más complejas. Las relaciones de recurrencia aparecen frecuentemente cuando se trata de determinar las características de rendimiento de programas recursivos y así se verán varios ejemplos a lo largo de este libro. Por ejemplo, en el Capítulo 9 aparece la ecuación:

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}), \quad \text{para } N \geq 1 \text{ con } C_0 = 1.$$

La manera más fácil de calcular el valor de C_N es utilizar un array, como en el programa anterior. En el Capítulo 9 se presentará cómo puede resolverse matemáticamente esta fórmula y en el Capítulo 6 se tratarán otras recurrencias distintas que suelen presentarse en el análisis de algoritmos.

Así, con frecuencia la relación entre programas recursivos y funciones definidas recursivamente es más filosófica que práctica. Hablando estrictamente, los problemas antes indicados no se asocian con el concepto de recursión en sí mismo, sino con la implementación: un compilador (muy inteligente) puede descubrir que la función factorial podría en realidad implementarse con un bucle y que la función Fibonacci se emplea mejor cuando se almacenan todos los valores precalculados en un array. Posteriormente se examinarán con más detalle los mecanismos de implementación de programas recursivos.

Divide y vencerás

La mayor parte de los programas recursivos que se consideran en este libro utilizan dos llamadas recursivas, y cada una opera sobre aproximadamente la mitad de los datos de entrada. Éste es el paradigma de diseño de algoritmos denominado «divide y vencerás», que se emplea a menudo para obtener importantes mejoras. Los programas del tipo de divide y vencerás normalmente no se reducen a bucles triviales, como el programa anterior que calculaba el factorial, porque tienen dos llamadas recursivas y por lo regular no obligan a realizar un número excesivo de cálculos, como en el programa para los números de Fibonacci, expuesto anteriormente, porque los datos de entrada se dividen sin recubrimientos.

Como ejemplo, considérese la tarea de trazar las marcas de las pulgadas de una regla: existe una marca en el punto $1/2''$, marcas algo más cortas en los intervalos de $1/4''$, marcas todavía más cortas en los intervalos de $1/8''$, etc., como se muestra (de forma ampliada) en la Figura 5.1. Éste es un prototipo de los cálculos sencillos que realiza el método de divide y vencerás; posteriormente se verá que hay muchas formas de llevar a cabo esta tarea.

Si la resolución deseada es $1/2'''$, se efectúa un cambio de escala de manera que la tarea sea poner una marca en cada punto entre 0 y $2''$, sin incluir los puntos extremos. Se supone la existencia de un procedimiento `marcar(x, h)` para hacer una marca de h unidades de altura en la posición x . La marca central debe

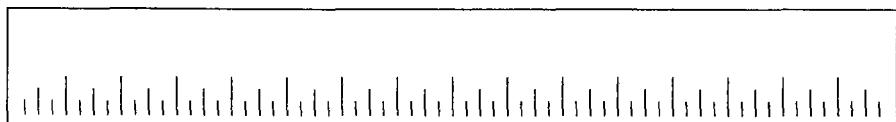


Figura 5.1 Una regla.

ser de n unidades de altura, las situadas en medio de las partes izquierda y derecha deben ser de $n - 1$ unidades de altura, etc. El siguiente programa recursivo de «divide y vencerás» constituye una forma directa de lograr el objetivo:

```
regla (int izq, int der, int h)
{
    int m = (izq+der)/2;
    if (h > 0)
    {
        marcar(m, h);
        regla(izq, m, h-1);
        regla(m, der, h-1);
    }
}
```

Por ejemplo, la llamada a `regla(0, 64, 6)` producirá la Figura 5.1, con la escala apropiada. El método se basa en la siguiente idea: para hacer las marcas de un intervalo, se comienza por la más grande, justo en el medio. Esto divide el intervalo en dos partes iguales. A continuación se hacen las marcas (más cortas) de cada mitad, utilizando el mismo procedimiento.

Normalmente conviene prestar especial atención a la condición de terminación de un programa recursivo; ya que de otra manera ¡puede que no termine nunca! En el programa anterior, la instrucción `regla` termina (no se llama más a sí misma) cuando la altura de las marcas que quedan por hacer es 0. La Figura 5.2 muestra el proceso con detalle, y proporciona la lista de las llamadas al procedimiento y las marcas que resultan al llamar a `regla(0, 8, 3)`. Se coloca una marca en el medio y se llama `regla` para la mitad izquierda, repitiendo sucesivamente el proceso hasta que la altura de las marcas sea 0. Para concluir se vuelve a llamar a `regla` para hacer las marcas de la mitad derecha de la misma manera.

En este problema no es particularmente importante el orden en el que se dibujen las marcas. Se podría haber puesto también la llamada a `marcar` entre las dos llamadas recursivas, en cuyo caso los puntos del ejemplo se trazarían simplemente en el orden de izquierda a derecha, como muestra la Figura 5.3.

El conjunto de marcas dibujadas por estos dos procedimientos es el mismo, pero el orden es bastante diferente. Esta diferencia puede explicarse mediante el árbol del diagrama de la Figura 5.4. Este diagrama tiene un nodo para cada llamada a `regla`, con los parámetros utilizados en ella etiquetados; los hijos de cada uno de ellos corresponden a las llamadas (recursivas) a `regla`, junto con sus parámetros correspondientes. Un árbol de este tipo permite ilustrar las características dinámicas de cualquier conjunto de procedimientos. La Figura 5.2 corresponde al recorrido de este árbol en preorden (la «visita» a un nodo corresponde a hacer la correspondiente llamada a `marcar`); la Figura 5.3 corresponde al recorrido en orden simétrico.

```

regla(0,8,3)
  marcar(4,3)      [     |     ]
  regla(0,4,2)
    marcar(2,2)      [   |   ]
    regla(0,2,1)
      marcar(1,1)      [   |   ]
      regla(0,1,0)
      regla(1,2,0)
    regla(2,4,1)
      marcar(3,1)      [   |   |   ]
      regla(2,3,0)
      regla(3,4,0)

regla(4,8,2)
  marcar(6,2)      [   |   |   |   ]
  regla(4,6,1)
    marcar(5,1)      [   |   |   |   |   ]
    regla(4,5,0)
    regla(5,6,0)
  regla(6,8,1)
    marcar(7,1)      [   |   |   |   |   |   ]
    regla(6,7,0)

```

Figura 5.2 Dibujo de una regla.

En general, los algoritmos del tipo divide y vencerás incluyen algún tratamiento para dividir los datos de entrada en dos partes, o para mezclar los resultados del proceso de la «resolución» de dos partes independientes de datos de entrada, o para hacer una rehabilitación después de que se haya procesado la mitad de los datos de entrada. Es decir, se pueden encontrar instrucciones antes, después o entre las dos llamadas recursivas. Más adelante se verán muchos ejemplos de este tipo de algoritmos, en especial en los Capítulos 9, 12, 27, 28 y 41. También se encontrarán algoritmos en los que no será posible aplicar completamente el método de divide y vencerás. Esto ocurre, por ejemplo, cuando los datos de entrada están divididos en dos partes desiguales, o están divididos en más de dos partes, o existe algún solapamiento entre las partes.

También es fácil desarrollar algoritmos no recursivos para esta tarea. El método más directo consiste simplemente en dibujar las marcas en orden, como en la Figura 5.3, pero con el bucle directo `for (i = 1; i < N; i++)` `marcar (i, altura(i));`. Se necesita la función `altura(i)` que no es difícil de calcular: es el número de bits consecutivos al final de la representación binaria de `i`. Se deja como ejercicio para el lector la implementación de esta función en

```

regla(0,8,3)
  regla(0,4,2) .
    regla(0,2,1)
      regla(0,2,1)
        regla(0,1,0)
        marcar(1,1)   [  ]
        regla(1,2,0)
        marcar(2,2)   [  ]
      regla(2,4,1)
        regla(2,3,0)
        marcar(3,1)   [  ]
        regla(3,4,0)
        marcar(4,3)   [  ]
      regla(4,8,2)
        regla(4,6,1)
          regla(4,5,0)
          marcar(5,1)   [  ]
          regla(5,6,0)
          marcar(6,2)   [  ]
        regla(6,8,1)
          regla(6,7,0)
          marcar(7,1)   [  ]

```

Figura 5.3 Dibujo de una regla (versión en orden simétrico).

C++. De hecho, es posible obtener este método directamente a partir de la versión recursiva mediante un proceso laborioso de «eliminación de la recursión», que se examinará con detalle más adelante, para otro problema.

Otro algoritmo no recursivo, que no corresponde a ninguna implementación recursiva, consiste en dibujar primero las marcas más cortas, luego las más

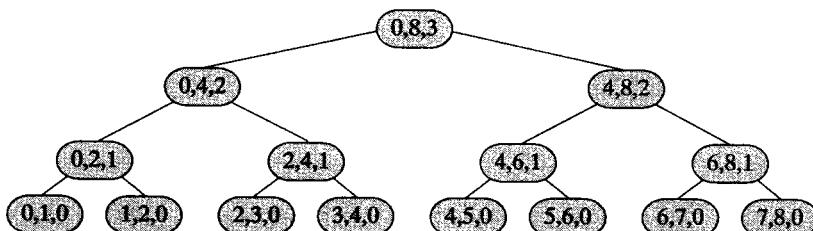


Figura 5.4 Árbol de llamada recursiva para dibujar una regla.

cortas de las restantes y así sucesivamente, como lo hace el pequeño programa siguiente:

```
regla (int izq, int der, int h)
{
    int i, j, t;
    for (i = 1, j = 1; i <=h; i++, j+=j)
        for (t = 0; t <= (izq+der)/j; t++)
            marcar(izq+j+t*(j+j), i);
}
```

La Figura 5.5 muestra cómo dibuja las marcas este programa. El proceso corresponde a recorrer el árbol de la Figura 5.4 en orden de nivel (de abajo hacia arriba), pero no es recursivo.

Este proceso corresponde al método general de diseño de algoritmos en el que se resuelve un problema tratando primero subproblemas triviales y combiniando después las soluciones para resolver subproblemas ligeramente más grandes, y así sucesivamente hasta la resolución de todo el problema. Esta manera

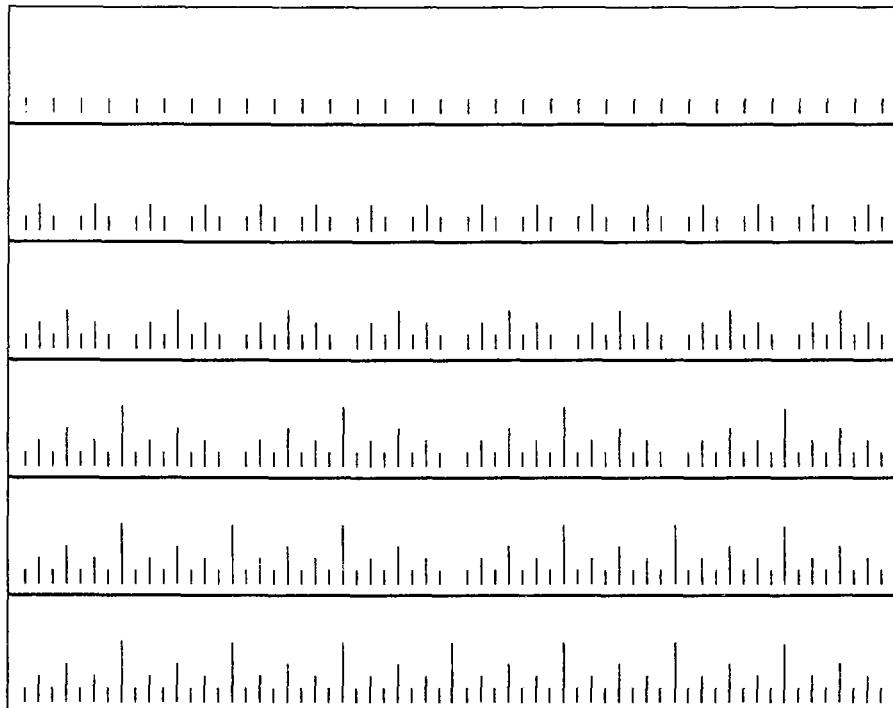


Figura 5.5 Dibujo no recursivo de una regla.

de trabajar podría denominarse «combina y vencerás». Mientras que todo programa recursivo admite una implementación no recursiva equivalente, no siempre es posible volver a ordenar los cálculos de esta forma —muchos programas recursivos dependen del orden específico en el que se resuelven los subproblemas—. Ésta es una aproximación *ascendente*, opuesta al método de divide y vencerás, en el que el orden de resolución es *descendente*. A lo largo del libro se encontrarán varios ejemplos de esto: el más importante en el Capítulo 12. En el Capítulo 42 se presenta una generalización del método.

Se ha examinado con detalle el ejemplo de dibujar una regla porque ilustra las propiedades esenciales de los algoritmos prácticos con una estructura similar a la de aquellos que se encontrarán más adelante. Con la recursión está justificado el estudio en profundidad de ejemplos simples, porque no es fácil saber cuándo se ha cruzado la frontera entre lo muy simple y lo muy complicado. La Figura 5.6 muestra un modelo bidimensional que ilustra cómo una descripción recursiva simple puede conducir a cálculos bastante complejos. El modelo de la izquierda tiene una estructura en la que se reconoce fácilmente su carácter recursivo, mientras que el modelo de la derecha parece más complicado si aparece en solitario, sin la compañía del primero. El programa que genera el modelo de la izquierda es, en realidad, una ligera generalización de regla:

```
estrella(int x, int y, int r)
{
    if (r > 0)
    {
        estrella(x-r,y+r,r/2);
        estrella(x+r,y+r,r/2);
        estrella(x-r,y-r,r/2);
        estrella(x+r,y-r,r/2);
        cuadrado(x,y,r);
    }
}
```

La primitiva de dibujo que se utiliza es simplemente un programa que dibuja un cuadrado de tamaño $2r$ y de centro (x, y) . Así, el modelo de la izquierda de la Figura 5.6 se genera de manera simple con un programa recursivo —el lector se puede entretenar intentando encontrar un método recursivo para dibujar el contorno del modelo de la derecha—. El modelo de la izquierda también es fácil de generar con un método ascendente como el que se representa en la Figura 5.5: se dibujan los cuadrados más pequeños, después los más pequeños de los restantes, etc. También puede ser interesante intentar encontrar un método no recursivo para dibujar el contorno.

Los modelos geométricos definidos recursivamente como los de la Figura 5.6 se denominan a veces *fractales*. Si se utilizan primitivas de dibujos más complejos e invocaciones recursivas más complicadas (en especial con funciones defi-

nidas recursivamente en los planos real y complejo), se pueden desarrollar modelos de gran complejidad y diversidad.

Recorrido recursivo de un árbol

Como se indicó en el Capítulo 4, el método más simple para recorrer los nodos de un árbol es probablemente con una implementación recursiva. Por ejemplo, el siguiente programa visita los nodos de un árbol binario en orden.

```
recorrer (struct nodo *t)
{
    if (t != z)
    {
        recorrer (t->izq);
        visitar (t);
        recorrer (t->der);
    }
}
```

La implementación refleja de forma precisa la definición del orden simétrico: «si el árbol no está vacío, recorrer primero el subárbol izquierdo, visitar la raíz y después recorrer el subárbol derecho». Evidentemente, el recorrido en orden previo puede implementarse poniendo la llamada a `visitar` antes de las dos llamadas recursivas, y el recorrido en orden posterior se puede implementar poniendo la llamada a `visitar` después de las dos llamadas recursivas.

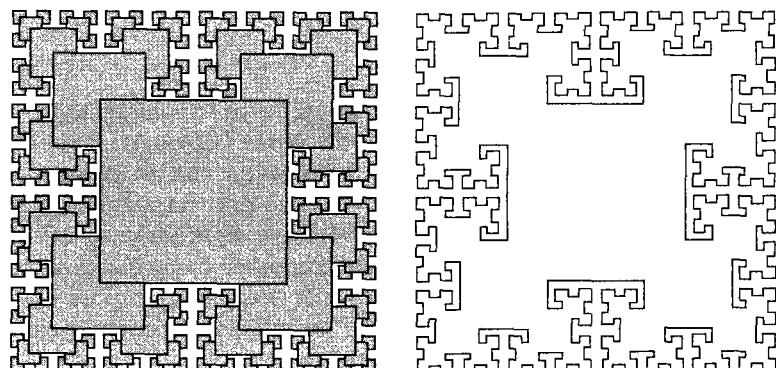


Figura 5.6 Una estrella fractal, dibujada con cuadrados (izquierda) y sólo con contornos (derecha).

Esta implementación recursiva del recorrido del árbol surge de una forma más natural que una implementación basada en una pila, ya que los árboles son estructuras definidas recursivamente y porque los recorridos en orden previo, en orden y en orden posterior son procesos definidos recursivamente. En contraste, se observa que no existe una forma adecuada de implementar un procedimiento recursivo para el recorrido en orden de nivel: la misma naturaleza de la recursión obliga a que los subárboles se procesen como unidades independientes, mientras que el orden de nivel necesita que los nodos de diferentes subárboles se mezclen entre ellos. Se volverá sobre este punto en los Capítulos 29 y 30, cuando se consideren los algoritmos de recorridos de grafos, que son estructuras mucho más complicadas que los árboles.

Unas simples modificaciones al programa recursivo anterior y la implementación apropiada de `visitar` pueden dar lugar a programas que calculen diversas propiedades de los árboles binarios de las figuras de este libro. Supóngase que el registro de los nodos incluye dos campos enteros para las coordenadas x y y de cada nodo en la página. (Para evitar detalles de escala y traslación, se supone que son coordenadas relativas: si el árbol tiene N nodos y es de altura h , la coordenada x va de izquierda a derecha desde 1 a N , y la coordenada y va desde arriba hacia abajo desde 1 a h .) El siguiente programa rellena estos campos con los valores apropiados para cada nodo:

```
visitar (struct nodo *t)
{
    t->x = ++x; t->y = y;
}
recorrer (struct nodo *t)
{
    y++;
    if (t != z)
    {
        recorrer (t->izq);
        visitar (t);
        recorrer (t->der)
    }
    y--;
}
```

El programa utiliza dos variables globales, x y y , que se suponen inicializadas a 0. La variable x sigue la pista del número de los nodos que se han visitado en orden; la variable y sigue la altura del árbol. Cada vez que `recorrer` desciende por el árbol, esta variable se incrementa en una unidad, y cada vez que asciende por el árbol, se disminuye también en una unidad.

De forma similar se podrían implementar programas recursivos para calcular la longitud del camino de un árbol, para encontrar otra forma de dibujar un árbol o para evaluar una expresión que él representa, etcétera.

Eliminación de la recursión

Pero, ¿cuál es la relación entre la implementación anterior (recursiva) y la implementación del Capítulo 4 (no recursiva) para el recorrido del árbol? Sin duda estos dos programas están fuertemente relacionados, ya que, para todo árbol dado, producen precisamente la misma serie de llamadas a visitar. En esta sección se estudia esta cuestión de forma detallada mediante la eliminación «mecánica» de la recursión del programa de recorrido en orden previo dado anteriormente para conseguir una implementación no recursiva.

Ésta es la misma operación con la que se enfrenta un compilador cuando tiene que traducir un programa recursivo a lenguaje de máquina. El objetivo principal de este apartado no es estudiar las técnicas de compilación (aunque se obtengan algunos conocimientos sobre los problemas con los que se enfrenta un compilador), sino más bien la relación entre las implementaciones recursivas y no recursivas de los algoritmos. Este tema se planteará nuevamente a lo largo del libro.

Para empezar se comienza con una implementación recursiva de recorrido en orden previo, exactamente como la antes descrita:

```
recorrer (struct nodo *t)
{
    if (t != z)
    {
        visitar (t);
        recorrer (t->izq);
        recorrer (t->der);
    }
}
```

La segunda llamada recursiva no va seguida de ninguna instrucción, por lo que puede eliminarse fácilmente. Cuando se va a ejecutar la segunda llamada, se invoca a recorrer (con el argumento t -> der); al finalizar esta llamada, se termina también la invocación *actual* de recorrer. Pero esta misma secuencia de acontecimientos se puede implementar con un goto en lugar de utilizar una llamada recursiva, como se indica a continuación:

```
recorrer (struct nodo *t)
{
    izq: if (t == z) goto x;
        visitar (t);
        recorrer (t->izq);
        t= t->der;
```

```

        goto izq;
x: ;
}

```

Ésta es una técnica muy conocida, denominada *eliminación de la recursión final*, que se implementa en muchos compiladores. Los programas recursivos son menos viables en los sistemas que no tienen esta capacidad, porque pueden aparecer aberraciones innecesarias y notables, tales como las producidas en las funciones factorial y fibonacci que se vieron anteriormente. En el Capítulo 9 se estudiará un ejemplo práctico importante.

La eliminación de la otra llamada recursiva requiere más trabajo. En general, la mayoría de los compiladores producen un código de instrucciones que sigue la misma secuencia de acciones en *cualquier* llamada de procedimiento: «colocar los valores de las variables locales y la dirección de la siguiente instrucción en la pila, definir los valores de los parámetros del procedimiento e (ir) goto al principio del procedimiento.» Entonces, cuando termina el procedimiento, se debe «sacar de la pila los valores y la dirección de retorno de las variables locales, inicializar las variables e (ir) goto a la dirección de retorno». Por supuesto, las cosas son más complejas en la mayoría de las situaciones que encuentra un verdadero compilador; no obstante, siguiendo en esta línea de trabajo, se puede eliminar la segunda llamada recursiva del programa de la siguiente forma:

```

recorrer (struct nodo *t)
{
    izq: if (t == z) goto s;
        visitar (t);
        pila.meter(t); t = t->izq; goto izq;
    der: t = t->der; goto izq;
    s:   if (pila.vacia ()) goto x;
        t = pila.sacar (); goto der;
    x: ;
}

```

Existe sólo una variable local t, por lo que se le introduce en la pila y se efectúa un goto al principio. Hay una única dirección de retorno der, que está fija, y por tanto no se pone en la pila. Al final del procedimiento, se actualiza t a partir del valor de la pila y se hace goto a la dirección de retorno der. Cuando la pila esté vacía, se retorna desde la primera llamada a recorrer.

Ahora ya se ha eliminado la recursión, pero se ha obtenido un «pantano» de gotos que integran un programa más bien opaco. Pero éstos se pueden eliminar «mecánicamente» para obtener un código más estructurado. En primer lugar, la parte de código comprendida entre la etiqueta der y el segundo goto x está rodeada de gotos y se puede mover fácilmente, y así se eliminan la etiqueta der y el goto asociado. Además, se observa que se asigna el valor t a t->der

cuando se saca de la pila; sería mejor meter ese valor en la pila. Finalmente, las instrucciones comprendidas entre la etiqueta x y el primer goto x no son más que un bucle while. Esto queda así:

```
recorrer (struct nodo *t)
{
    izq: while (t != z)
    {
        visitar (t);
        pila.meter(t->der); t = t->izq;
    }
    if (pila.vacia ()) goto x;
    t = pila.sacar (); goto izq;
x: ;
}
```

Ahora se tiene otro bucle, que se puede transformar en otro while añadiendo un apilamiento extra (del argumento inicial t a la entrada de recorrer), quedando un programa sin goto:

```
recorrer (struct nodo *t)
{
    pila.meter(t);
    while (!pila.vacia())
    {
        t = pila.sacar();
        while (t != z)
        {
            visitar (t) ;
            pila.meter (t->izq);
            t = t->l
        }
    }
}
```

Esta versión es el método de recorrido no recursivo «estándar». Es un ejercicio instructivo para olvidar por el momento cómo se obtuvo y para convencerse directamente de que este programa hace el recorrido del árbol en orden previo como se aconsejó.

En realidad, la estructura de este programa, con un bucle dentro de otro, puede simplificarse (con el coste de algunas inserciones en la pila):

```
recorrer (struct nodo * t)
```

```

{
    pila.meter(t);
    while (!pila.vacia())
    {
        t = pila.sacar();
        if (t != z)
        {
            visitar (t) ;
            pila.meter (t->der);
            pila.meter (t->izq);
        }
    }
}

```

Este programa tiene un parecido notable con el algoritmo recursivo original en orden previo, aunque en realidad los dos programas son *completamente* diferentes. Una primera diferencia es que este programa puede ejecutarse en prácticamente cualquier entorno de programación, mientras que, claro está, la implementación recursiva necesita un entorno que dé cabida a la recursión. Incluso en tal entorno, es probable que este método basado en pila sea algo más eficaz.

Finalmente, se observa que este programa coloca en la pila subárboles vacíos, como consecuencia de la decisión tomada en la implementación original de verificar que el subárbol no esté vacío como primera acción del procedimiento recursivo. La implementación recursiva podría hacer la llamada recursiva sólo para los subárboles no vacíos verificando $t \rightarrow izq$ y $t \rightarrow der$. Reflejar este cambio en el programa anterior conduce al algoritmo basado en pila para el recorrido en orden previo del Capítulo 4.

```

recorrer (struct nodo *t)
{
    pila.meter (t);
    while (!pila.vacia ())
    {
        t = pila.sacar(); visitar (t);
        if (t->der != z) pila.meter (t->der);
        if (t->izq != z) pila.meter (t->izq);
    }
}

```

Cualquier algoritmo recursivo puede manipularse de la manera precedente para eliminar la recursión; desde luego, ésta es la tarea principal del compilador. La eliminación «manual» de la recursión que se acaba de describir, aunque es

complicada, conduce frecuentemente a una implementación no recursiva eficaz y a un mejor entendimiento de la naturaleza de la operación.

Perspectiva

Seguramente es imposible hacer justicia a un tema tan fundamental como la recursión en una exposición tan breve. A lo largo del libro aparecen muchos de los mejores ejemplos de programas recursivos —se han ideado algoritmos del tipo de divide y vencerás para una amplia variedad de problemas—. En muchas aplicaciones no hay razón para ir más allá de una implementación recursiva, directa y simple; en otras, en cambio, se considerará la posibilidad de eliminar la recursión como se describió en este capítulo o se intentará obtener alguna alternativa de implementaciones no recursivas directamente.

La recursión está en el corazón de los primeros estudios teóricos sobre la verdadera naturaleza del cálculo informático. Los programas y funciones recursivos desempeñan un papel central en los estudios matemáticos que intentan separar los problemas que se pueden resolver mediante una computadora de los que no se pueden.

En el Capítulo 44 se estudiará la utilización de programas recursivos (y otras técnicas) para resolver problemas difíciles, en los que debe examinarse un gran número de posibles soluciones. Como se verá, la programación recursiva puede ser un medio bastante eficaz para organizar una búsqueda compleja en el conjunto de soluciones posibles.

Ejercicios

1. Escribir un programa recursivo para dibujar un árbol binario de manera que la raíz aparezca en el centro de la página, la raíz del subárbol izquierdo esté en el centro de la mitad izquierda de la página, etcétera.
2. Escribir un programa recursivo para calcular la longitud del camino externo de un árbol binario.
3. Escribir un programa recursivo para calcular la longitud del camino externo de un árbol representado como un árbol binario.
4. Obtener las coordenadas generadas cuando se aplica el procedimiento recursivo de dibujo del árbol dado en el texto al árbol binario de la Figura 4.2.
5. Eliminar mecánicamente la recursión del programa fibonacci dado en el texto, para obtener una implementación no recursiva.
6. Eliminar mecánicamente la recursión del algoritmo recursivo de recorrido del árbol *en orden*, para obtener una implementación no recursiva.

7. Eliminar mecánicamente la recursión del algoritmo recursivo de recorrido del árbol en *orden posterior* para obtener una implementación no recursiva.
8. Escribir un programa recursivo del tipo «divide y vencerás» para dibujar una aproximación del segmento que conecta dos puntos (x_1, y_1) y (x_2, y_2) , dibujando sólo los puntos que utilizan coordenadas enteras. (Pista: dibujar primero un punto próximo a la mitad.)
9. Escribir un programa recursivo para resolver el problema de Josefo (ver Capítulo 3).
10. Escribir una implementación recursiva del algoritmo de Euclides (ver Capítulo 2).

Análisis de algoritmos

Para la mayoría de los problemas existen varios algoritmos diferentes. ¿Cómo elegir uno que conduzca a la mejor implementación? Esta cuestión constituye actualmente un área de estudio muy desarrollada de la informática. Con frecuencia se tendrá la oportunidad de investigar los resultados que describen el comportamiento de algoritmos fundamentales. De cualquier forma, la comparación de algoritmos puede ser un desafío, por lo que serán útiles ciertas pautas generales.

Normalmente los problemas a resolver tienen un «tamaño» natural (en general, la cantidad de datos a procesar), al que se denominará N y en función del cual se tratará de describir los recursos utilizados (con frecuencia, la cantidad de tiempo empleado). El punto de interés es el estudio del *caso medio*, es decir, el tiempo de ejecución de un conjunto «tipo» de datos de entrada, y el del *peor caso*, el tiempo de ejecución para la configuración de datos de entrada más desfavorable.

Algunos de los algoritmos de este libro se entienden muy bien, hasta el punto de que se conocen las fórmulas matemáticas precisas para averiguar el tiempo de ejecución medio y el tiempo de ejecución del peor caso. Estas fórmulas se obtienen estudiando cuidadosamente el programa, para encontrar el tiempo de ejecución en términos de expresiones matemáticas fundamentales y a continuación hacer un análisis matemático de las cantidades implicadas. Por otra parte, las propiedades del rendimiento de otros algoritmos de este libro son totalmente desconocidas —quizás porque su análisis conduce a cuestiones matemáticas no resueltas, o porque se sabe que las implementaciones son demasiado complejas para analizarlas al detalle de una manera razonable, o (la mayoría de las veces) porque los tipos de entrada que se encuentran no pueden caracterizarse adecuadamente—. La mayoría de los algoritmos caen entre estos extremos: se conocen algunos hechos sobre su rendimiento, pero en realidad no se han llegado a analizar por completo.

Varios de los factores importantes que entran en este análisis habitualmente no son competencia del programador. En primer lugar, los programas en C++

se traducen a código de máquina para una computadora dada, y puede ser una tarea difícil averiguar exactamente cuánto se tarda en tratar incluso una sola sentencia de C++ (especialmente en un entorno donde se comparten los recursos de modo que el mismo programa pueda tener distintas características de rendimiento). En segundo lugar, muchos programas son excesivamente sensibles a sus datos de entrada, y su rendimiento podría fluctuar enormemente según sean éstos. El caso medio podría ser una ficción matemática que no es representativa de los datos reales que utiliza cada programa, y el peor caso podría ser una construcción rara que nunca ocurriría en la práctica. En tercer lugar, muchos programas de interés no se entienden bien, y puede que no proporcionen los resultados matemáticos específicos. Por último, es frecuente el caso en el que los programas no sean comparables en absoluto: uno es mucho más rápido que otro para un tipo particular de entrada, y el otro lo es bajo otras circunstancias.

A pesar de los comentarios anteriores, a menudo es posible predecir con exactitud el tiempo de ejecución de un programa particular o saber que un programa funcionará mejor que otro en situaciones concretas. El objetivo del analista de algoritmos es descubrir tanta información como sea posible sobre el desarrollo de los algoritmos; la tarea del programador es aplicar esta información para seleccionar los algoritmos para cada aplicación en particular. En este capítulo, el centro de atención será el mundo más bien idealizado del analista; en el siguiente se estudiarán consideraciones prácticas de implementación.

Marco de referencia

El primer paso del análisis de un algoritmo es establecer las características de los datos de entrada que utilizará y decidir cuál es el tipo de análisis más apropiado. Idealmente, sería deseable poder obtener, para cualquier distribución de probabilidad de las posibles entradas, la correspondiente distribución de los tiempos empleados en la ejecución del algoritmo. Desgraciadamente no es posible alcanzar este ideal para un algoritmo que no sea trivial, de manera que, por lo regular, se limita el desarrollo estadístico intentando probar que el tiempo de ejecución es siempre menor que algún «límite superior» *sea cual sea la entrada*, e intentando obtener el tiempo de ejecución *medio* para su entrada «aleatoria».

El segundo paso del análisis de un algoritmo es identificar las operaciones abstractas en las que se basa, con el fin de separar el análisis de la implementación. Así, por ejemplo, se separa el estudio del número de comparaciones que realiza un algoritmo de ordenación del estudio para determinar cuántos microsegundos tarda una computadora concreta en ejecutar un código de máquina cualquiera producido por un compilador determinado para el fragmento de código `if (a[i] < v) ...`. Ambos casos se necesitan para determinar cuál es el tiempo de ejecución real del programa en una computadora en particular. El

primero dependerá de las propiedades del algoritmo, mientras que el segundo dependerá de las propiedades de la computadora. Esta separación permite a menudo comparar algoritmos, independientemente de las implementaciones particulares o de las computadoras que se puedan utilizar.

Mientras que el número de operaciones abstractas implicadas puede ser, en principio, grande, normalmente se da el caso de que el desarrollo de los algoritmos que se consideran depende sólo de unas cuantas cantidades. En general, es fácil identificar las cantidades significativas para un programa en particular —una forma de hacerlo consiste en utilizar una opción de «detección de perfiles» (disponible en muchas implementaciones de C++) para realizar estadísticas de la frecuencia de llamada de una instrucción en algún ejemplo de ejecución—. En este libro, el interés se centra en las cantidades de ese tipo que sean importantes para cada programa.

El tercer paso del análisis de un algoritmo es analizarlo matemáticamente, con el fin de encontrar los valores del caso medio y del peor caso para cada una de las cantidades fundamentales. No es difícil encontrar un límite superior del tiempo de ejecución de un programa —el reto es encontrar el *mejor* límite superior, aquel que se encontraría si se diera la peor entrada posible—. Esto produce el peor caso: el caso medio normalmente requiere un análisis matemático más sofisticado. Una vez desarrollados con éxito tales análisis para las cantidades fundamentales, se puede determinar el tiempo asociado a cada cantidad y obtener expresiones para el tiempo total de ejecución.

En principio, el rendimiento de un algoritmo se puede analizar a menudo con un nivel de precisión detallado, limitado sólo por la incertidumbre sobre el rendimiento de la computadora o por la dificultad de determinar las propiedades matemáticas de algunas de las cantidades abstractas. Sin embargo, rara vez será útil hacer un análisis detallado completo, de manera que siempre será preferible una *estimación* mejor que un cálculo preciso. (En realidad, las estimaciones que aparentemente son sólo aproximadas, a menudo resultan ser bastante precisas.) Tales estimaciones aproximadas se obtienen fácilmente por medio del viejo refrán del programador: «el 90 % del tiempo se emplea en el 10 % de la codificación.» (En el pasado ya se decía esto pero con otros valores diferentes del «90 %».)

El análisis de un algoritmo es un proceso cíclico, estimándolo y refinándolo hasta que se alcanza una respuesta al nivel de precisión deseado. Realmente, como se estudiará en el siguiente capítulo, el proceso también debería incluir mejoras en la implementación y desde luego el análisis sugiere a menudo tales mejoras.

Teniendo en cuenta estas advertencias, el *modo de proceder* será buscar estimaciones aproximadas del tiempo de ejecución de los programas con el fin de clasificarlos, sabiendo que se puede hacer un análisis más completo para programas importantes cuando sea necesario.

Clasificación de los algoritmos

Como se mencionó antes, la mayoría de los algoritmos tienen un parámetro primario N , normalmente el número de elementos de datos a procesar, que afecta muy significativamente al tiempo de ejecución. El parámetro N podría ser el grado de un polinomio, el tamaño de un archivo a ordenar o en el que se va a realizar una búsqueda, el número de nodos de un grafo, etc. Prácticamente todos los algoritmos de este libro tienen un tiempo de ejecución proporcional a una de las siguientes funciones:

- 1 La mayor parte de las instrucciones de la mayoría de los programas se ejecutan una vez o muy pocas veces. Si todas las instrucciones de un programa tienen esta propiedad, se dice que su tiempo de ejecución es *constante*. Obviamente, esto es lo que se persigue en el diseño de algoritmos.
- $\log N$ Cuando el tiempo de ejecución de un programa es *logarítmico*, éste será ligeramente más lento a medida que crezca N . Este tiempo de ejecución es normal en programas que resuelven un problema de gran tamaño transformándolo en uno más pequeño, dividiéndolo mediante alguna fracción constante. Para lo que aquí interesa, el tiempo de ejecución puede considerarse menor que una «gran» constante. La base del logaritmo cambia la constante, pero no mucho: cuando N vale mil, si la base es 10, $\log N$ es 3 y si la base es 2 es aproximadamente 10; cuando N vale un millón, $\log N$ se multiplica por dos. Cuando se dobla N , $\log N$ crece de forma constante, pero no se duplica hasta que N llegue a N^2 .
- N Cuando el tiempo de ejecución de un programa es *lineal*, eso significa generalmente que para cada elemento de entrada se realiza una pequeña cantidad de procesos. Cuando N vale un millón, este valor es también el del tiempo de ejecución, que se duplica al hacerlo N . Ésta es la situación ideal para un algoritmo que debe procesar N entradas (u obtener N salidas).
- $N \log N$ Este tiempo de ejecución es el de los algoritmos que resuelven un problema dividiéndolo en pequeños subproblemas, resolviéndolos independientemente, y combinando después las soluciones. Ante la falta de un adjetivo mejor (*lineal-aritmético?*), se dice que el tiempo de ejecución de tal algoritmo es « $N \log N$ ». Cuando N vale un millón $N \log N$ es aproximadamente veinte millones. Cuando se duplica N , el tiempo de ejecución es más del doble (aunque no mucho más).
- N^2 Cuando el tiempo de ejecución de un algoritmo es *cuadrático*, sólo es práctico para problemas relativamente pequeños. El tiempo de ejecución cuadrático normalmente aparece en algoritmos que procesan

pares de elementos de datos (por ejemplo, en un bucle anidado doble). Cuando N vale mil, el tiempo de ejecución es un millón. Cuando N se dobla, el tiempo de ejecución se multiplica por cuatro.

- N^3 De igual manera, un algoritmo que procesa tríos de elementos de datos (por ejemplo, en un bucle anidado triple) tiene un tiempo de ejecución *cúbico* y no es útil más que en problemas pequeños. Cuando N vale cien, el tiempo de ejecución es un millón. Cuando N se duplica, el tiempo de ejecución se multiplica por ocho.
- 2^N Pocos algoritmos con un tiempo de ejecución *exponencial* son susceptibles de poder ser útiles en la práctica, aunque aparecen de forma natural al aplicar el método de la «fuerza bruta» en la resolución de problemas. Cuando N vale veinte, el tiempo de ejecución es un millón. Cuando N dobla su valor, el tiempo de ejecución se eleva al cuadrado!

El tiempo de ejecución de un programa particular es probablemente igual a alguna constante multiplicada por uno de sus términos (el «término principal») más algunos términos más pequeños. Los valores del coeficiente constante y de los términos incluidos dependen de los resultados del análisis y de los detalles de la implementación. De forma esquemática, el coeficiente del término principal está condicionado por el número de instrucciones que hay en el bucle interno: en cualquier nivel del diseño del algoritmo, es prudente limitar el número de estas instrucciones. Para grandes valores de N se impone el efecto del término principal; para pequeños valores de N o para algoritmos diseñados minuciosamente pueden contribuir más términos, y la comparación de algoritmos es más difícil. En la mayoría de los casos, simplemente se dice que el tiempo de ejecución de los programas es «lineal», « $N \log N$ », «cúbico», etc., entendiéndose implícitamente que en los casos donde sea muy importante la eficacia, debe realizarse un análisis más detallado o un estudio empírico.

A veces surgen otras funciones. Por ejemplo, un algoritmo con N^2 entradas que tiene un tiempo de ejecución cúbico en N es más adecuado clasificarlo como un algoritmo de tiempo de ejecución $N^{3/2}$. También, algunos algoritmos tienen dos etapas de descomposición en subproblemas, lo que se traduce en un tiempo de ejecución proporcional a $N \log^2 N$. Ambas funciones se aproximan mucho más a $N \log N$ que a N^2 , para valores grandes de N .

A continuación se ampliará lo antes dicho sobre la función « \log ». Como se mencionó, la base del logaritmo hace cambiar los cálculos de forma constante. Como frecuentemente se trata sólo con resultados analíticos con un factor constante, no importa mucho cuál es la base, por lo que se dice simplemente « $\log N$ », etc. Por otro lado, algunas veces se dará el caso de que los conceptos se explicarán más claramente si se utiliza alguna base específica. En matemáticas, el *logaritmo natural* (o *neperiano* en base $e = 2.718281828\dots$) aparece con tanta frecuencia que lo normal es utilizar una abreviatura especial: $\log_e N \equiv \ln N$. En informática, el *logaritmo binario* (base 2) aparece tan a menudo que se utiliza

$\lg N$	$\lg^2 N$	\sqrt{N}	N	$N\lg N$	$N\lg^2 N$	$N^{3/2}$	N^2
3	9	3	10	30	90	30	100
6	36	10	100	600	3.600	1.000	10.000
9	81	31	1.000	9.000	81.000	31.000	1.000.000
13	169	100	10.000	130.000	1.690.000	1.000.000	100.000.000
16	256	316	100.000	1.600.000	25.600.000	31.600.000	10 mil millones
19	361	1.000	1.000.000	19.000.000	361.000.000	mil millones	un billón

Figura 6.1 Valores relativos aproximados de funciones.

la notación abreviada $\log_2 N \equiv \lg N$. Por ejemplo, el mayor entero inferior a $\lg N$ indica el número de bits necesario para representar N en escritura binaria.

La Figura 6.1 indica el tamaño relativo de algunas de estas funciones: la tabla proporciona los valores aproximados de $\lg N$, $\lg^2 N$, \sqrt{N} , N , $N\lg N$, $N\lg^2 N$, $N^{3/2}$, N^2 para diferentes valores de N . La función cuadrática domina claramente, en especial para N grande, pero las diferencias entre las funciones más pequeñas no son las que podría esperarse para un N pequeño. Por ejemplo, $N^{3/2}$ debería ser mayor que $N\lg^2 N$ para N muy grande, pero no para los valores más pequeños que son los que podrían darse en la práctica. Se sobreentiende que no se da esta tabla para que se haga una comparación lineal de las funciones para todos los valores de N —números, tablas y grafos relativos a los algoritmos específicos pueden hacer mejor esta tarea—, pero proporciona una primera aproximación bastante realista.

Complejidad del cálculo

Una técnica de aproximación al estudio del rendimiento de los algoritmos consiste en examinar el *peor caso*, sin tener en cuenta los factores constantes, con el fin de determinar la dependencia funcional del tiempo de ejecución (o alguna otra medida) del número de datos de entrada (o de alguna otra variable). Este enfoque es interesante porque permite *demonstrar* propiedades matemáticas precisas sobre el tiempo de ejecución de los programas: por ejemplo, se puede afirmar que el tiempo de ejecución de una ordenación por mezcla (ver Capítulo 11) es *forzosamente* proporcional a $N\log N$.

El primer paso del proceso consiste en precisar matemáticamente lo que se entiende por «proporcional a», lo que al mismo tiempo separará el análisis de un algoritmo de cualquier implementación particular. La idea es ignorar los factores constantes en el análisis: en la mayor parte de los casos, si se desea conocer si el tiempo de ejecución de un algoritmo es proporcional a N o a $\log N$, no tiene importancia si el algoritmo se ejecutará en una microcomputadora o en una supercomputadora y tampoco si el bucle interno se ha implementado cuidadosamente, con sólo unas pocas instrucciones, o si por el contrario se ha

hecho mal, con muchas instrucciones. Desde un punto de vista matemático, estos dos factores son equivalentes.

El artificio matemático que permite precisar esta idea se denomina *notación O*, o «notación O mayúscula», definida de la siguiente forma:

Notación. Se dice que una función $g(N)$ pertenece a $O(f(N))$ si existen las constantes c_0 y N_0 tales que $g(N)$ es menor que $c_0 f(N)$ para todo $N > N_0$.

Informalmente, esto engloba la idea de «es proporcional a» y libera al analista de considerar los detalles de las características particulares de la máquina. Además, la afirmación de que el tiempo de ejecución de un algoritmo pertenece a $O(f(N))$ es independiente de los datos de entrada del algoritmo. Como el interés está en el estudio del *algoritmo*, no en sus datos de entrada o en la implementación, la notación O es una forma útil de encontrar un límite superior del tiempo de ejecución, independientemente de los detalles de la implementación y de los datos de entrada.

La notación O ha sido de gran utilidad para los analistas, al ayudar a clasificar algoritmos por su tiempo de ejecución, y también para los diseñadores, al orientar la búsqueda de los «mejores» algoritmos adecuados para problemas importantes. La meta del estudio del *cálculo de la complejidad* de un algoritmo es demostrar que su tiempo de ejecución pertenece a $O(f(N))$ para alguna función f , y que no puede haber ningún algoritmo con tiempo de ejecución en $O(g(N))$ para cualquier función $g(N)$ «inferior» (una función tal que $\lim_{N \rightarrow \infty} g(N)/f(N) = 0$). Se trata de proporcionar un «límite superior» y un «límite inferior» para el tiempo de ejecución del peor caso. El cálculo de los límites superiores consiste normalmente en analizar y determinar la frecuencia de las operaciones (se verán muchos ejemplos en los capítulos siguientes), mientras que el de los límites inferiores implica la difícil tarea de construir cuidadosamente un modelo de máquina y determinar qué operaciones fundamentales debe ejecutar cualquier algoritmo para resolver un problema (rara vez se tratará este punto). Cuando los cálculos teóricos indiquen que el límite superior de un algoritmo coincide con su límite inferior, entonces se tendrá la seguridad de que es inútil tratar de diseñar un algoritmo fundamentalmente más rápido y, por tanto, se puede dedicar toda la atención a la implementación. Este punto de vista ha resultado ser muy útil en el diseño de algoritmos en los últimos años.

Sin embargo, al utilizar la notación O hay que ser extremadamente cuidadoso al interpretar los resultados, al menos por cuatro razones: primera, es un «límite superior» y la cantidad en cuestión podría ser mucho menor; segunda, podría ocurrir que la entrada que provoca el peor caso no se dé nunca en la práctica; tercera, no se conoce la constante c_0 y no tiene por qué ser pequeña; y cuarta, también se desconoce la constante N_0 y tampoco tiene por qué ser pequeña. A continuación se considerarán estas razones, una a continuación de otra.

La afirmación de que el tiempo de ejecución de un algoritmo pertenece a $O(f(N))$ no implica que el algoritmo siempre tarde tanto: sólo dice que el ana-

N	$\frac{1}{4}N\lg^2 N$	$\frac{1}{2}N\lg^2 N$	$N\lg^2 N$	$N^{3/2}$
10	22	45	90	30
100	900	1.800	3.600	1.000
1.000	20.250	40.500	81.000	31.000
10.000	422.500	845.000	1.690.000	31.600.000
1.000.000	90.250.000	180.500.000	361.000.000	1.000.000.000

Figura 6.2 Importancia de los factores constantes en la comparación de funciones.

lista ha podido comprobar que nunca tardará más. El tiempo real de ejecución podría ser siempre muy inferior. Se ha desarrollado la mejor notación para cubrir la situación en la que se sabe que existe alguna entrada para la cual el tiempo de ejecución pertenece a $O(f(N))$, pero hay muchos algoritmos para los que resulta bastante complicado construir los datos de entrada del peor caso.

Aun cuando se conozca la entrada del peor caso, puede darse la situación en que los datos de entrada que se encuentren realmente en la práctica tengan tiempos de ejecución mucho más pequeños. Muchos algoritmos sumamente útiles tienen un mal comportamiento en el peor caso. Por ejemplo, el que probablemente sea el algoritmo de ordenación más extendido, el Quicksort, tiene un tiempo de ejecución en $O(N^2)$, pero es posible organizar los datos de forma que el tiempo de ejecución para las entradas que se encuentran en la práctica sea proporcional a $N \log N$.

Las constantes c_0 y N_0 implícitas en la notación O a menudo ocultan detalles de la implementación que son importantes en la práctica. Evidentemente, decir que un algoritmo tiene un tiempo de ejecución en $O(f(N))$ no proporciona ningún dato sobre el tiempo de ejecución si N es menor que N_0 y c_0 pudiera estar ocultando una gran cantidad de «valores superiores» diseñados para evitar un peor caso malo. Es preferible un algoritmo que utilice N_2 nanosegundos a otro que utilice $\log N$ siglos, pero no se puede hacer esta elección basándose en la notación O . La Figura 6.2 muestra la situación para dos funciones típicas, con valores de las constantes más realistas, en el intervalo $0 \leq N \leq 1.000.000$. La función $N^{3/2}$, que se podría haber considerado erróneamente como la mayor de las cuatro, ya que es asintóticamente la más grande, en realidad es de las más pequeñas para pequeños valores de N , y es menor que $N \lg^2 N$ hasta que N valga unas decenas de miles. Los programas en los que los tiempos de ejecución dependen de funciones de este tipo no se pueden comparar de forma eficaz sin prestar una atención cuidadosa a los factores constantes y a los detalles de la implementación.

Es conveniente pensar detenidamente dos veces antes de utilizar, por ejemplo, un algoritmo con tiempo de ejecución en $O(N^2)$ en lugar de uno en $O(N)$, pero tampoco se deben seguir ciegamente los resultados del cálculo de la complejidad expresados en notación O . En las implementaciones prácticas de los algoritmos considerados en este libro, la complejidad del cálculo es a veces demasiado general y la notación O demasiado imprecisa para ser útil. La complejidad del cálculo debe considerarse como el primer paso de un proceso

progresivo de refinamiento del análisis de un algoritmo, para dar a conocer más detalles sobre sus propiedades. En este libro se centra el interés en los pasos siguientes, más próximos a las implementaciones reales.

Análisis del caso medio

Otra forma de estudiar el rendimiento de los algoritmos consiste en examinar el *caso medio*. En la situación más simple, es posible caracterizar con precisión los datos de entrada del algoritmo: por ejemplo, un algoritmo de ordenación puede operar sobre un array de N enteros aleatorios o un algoritmo geométrico puede procesar un conjunto de N puntos aleatorios del plano con coordenadas entre 0 y 1. Entonces se calcula el número medio de veces que se ejecuta cada instrucción y se obtiene el tiempo medio de ejecución del programa multiplicando la frecuencia de cada instrucción por el tiempo que se necesita para dicha instrucción y sumando todas estas cantidades. Sin embargo, al hacer esto existen al menos tres dificultades, que se van a examinar una a una.

La primera es que en algunas computadoras puede resultar difícil determinar con precisión el tiempo que se necesita para cada instrucción. Peor aún, dicho tiempo está sujeto a cambios, y una gran parte de los análisis realizados en una computadora puede no tener valor para los tiempos de ejecución del mismo algoritmo en otra computadora. Éste es exactamente el tipo de problemas que trata de evitar el estudio de la complejidad del cálculo.

Segunda: a menudo el análisis del caso medio es en sí mismo un desafío matemático difícil que requiere argumentos detallados y complejos. Por su naturaleza, los cálculos matemáticos necesarios para comprobar los límites superiores son normalmente menos complejos porque no necesitan ser tan precisos. Todavía se desconoce el rendimiento del caso medio de muchos algoritmos.

Tercera, y la más importante: en el análisis del caso medio puede ser que el modelo de datos de entrada no caracterice con precisión a los que aparecen en la práctica, o puede ser que no exista ningún modelo de entrada natural. ¿Cómo se deberían caracterizar los datos de entrada de un programa de tratamiento de textos en inglés? Pero, al contrario, existen pocos argumentos contra la utilización de modelos de entrada tales como «un archivo ordenado aleatoriamente» para un algoritmo de ordenación, o «un conjunto de puntos aleatorios» para un algoritmo geométrico. Para tales modelos es posible obtener resultados matemáticos que permitan predecir con precisión el rendimiento de los programas que operan en aplicaciones reales. Aunque la obtención de estos resultados normalmente rebasa los objetivos de este libro, se presentan algunos ejemplos (ver Capítulo 9), y se mencionarán algunos resultados significativos cuando sea necesario.

Resultados aproximados y asintóticos

A menudo, los resultados de un análisis matemático no son exactos sino aproximados en un sentido técnico preciso: el resultado podría ser una expresión compuesta por una sucesión de términos decrecientes. De igual forma que se presta más atención al bucle interno de un programa, se está más interesado en el *término más significativo* (el término más grande) de una expresión matemática. La notación O se desarrolló originalmente para este tipo de aplicación, y, usada adecuadamente, permite realizar estimaciones concisas que dan buenas aproximaciones a resultados matemáticos.

Supóngase, por ejemplo (después de algunos análisis matemáticos), que se determina que un algoritmo particular tiene un bucle interno que está repetido MgN veces de media, que una sección externa lo está N veces y que algún código de inicialización se ejecuta una sola vez. Supóngase además que se descubre (después de un minucioso examen de la implementación) que cada iteración del bucle interior requiere a_0 microsegundos, las de la sección externa, a_1 microsegundos, y que la inicialización se hace en a_2 microsegundos. Entonces el tiempo medio de ejecución del programa (en microsegundos) es

$$a_0N \lg N + a_1N + a_2.$$

Pero también es cierto que el tiempo de ejecución es

$$a_0N \lg N + O(N).$$

(El lector puede verificar esta afirmación a partir de la definición de $O(N)$.) Esto es importante porque, si se está interesado en una respuesta aproximada, se sabe que, para N grande, puede no necesitarse encontrar los valores de a_1 o a_2 . Más importante aún, puede que otros términos de la expresión exacta del tiempo de ejecución sean difíciles de analizar: la notación O proporciona una forma de obtener una respuesta aproximada para valores de N suficientemente grandes sin preocuparse de tales términos.

Técnicamente, no se tiene una seguridad real de que se puedan ignorar los términos pequeños de esta manera, porque la definición de la notación O no dice nada sobre el tamaño de la constante c_0 que podría ser muy grande. Pero (aunque, por lo regular, no sea una preocupación) en tales casos hay formas para acotar las constantes que son pequeñas en comparación con N , y así normalmente está justificado ignorar las cantidades representadas por la notación O cuando existe un término principal (mayor) bien definido. Al hacer esto se tiene la seguridad de que se poseen los conocimientos necesarios para efectuar tal simplificación si fuera necesario, aunque raramente se hará así.

De hecho, cuando una función $f(N)$ sea asintóticamente grande comparada con otra función $g(N)$, se utilizará en este libro la terminología (decididamente

no técnica) «del orden de $f(N)$ » para significar $f(N) + O(g(N))$. De esta manera, lo que se pierde en precisión matemática se gana en claridad, ya que el interés radica más en el rendimiento de los algoritmos que en los detalles matemáticos. En tales casos, el lector puede estar seguro de que, para valores de N suficientemente grandes (si no es para todo N), la cantidad en cuestión estará muy próxima a $f(N)$. Por ejemplo, incluso si se sabe que una cantidad es $N(N - 1)/2$, se puede hacer referencia a ella como «del orden de» $N^2/2$. Esto se percibe más rápidamente y, por ejemplo, el error cometido es de un 10 % cuando $N = 1000$. La precisión perdida en tales casos es despreciable comparada con la que se pierde al utilizar $O(f(N))$. El objetivo es ser a la vez precisos y concisos en la descripción del rendimiento de los algoritmos.

Recurrencias básicas

Como se verá en los siguientes capítulos, un gran número de algoritmos se basan en el principio de descomponer recursivamente un problema grande en otros más pequeños, utilizando las soluciones de los subproblemas para resolver el problema original. El tiempo de ejecución de estos algoritmos viene determinado por el tamaño y el número de los subproblemas, así como por el coste de la descomposición. En esta sección se verán métodos básicos para analizar tales algoritmos y obtener soluciones de unas cuantas fórmulas estándar que aparecen en el análisis de muchos de los algoritmos que se estudiarán más adelante. La comprensión de las propiedades matemáticas de las fórmulas de esta sección permitirá delimitar las propiedades del rendimiento de los algoritmos de este libro.

La propia naturaleza de un programa recursivo impone que su tiempo de ejecución para una entrada de tamaño N dependa de su tiempo de ejecución para entradas más pequeñas: esto conduce de nuevo a las *relaciones de recurrencia*, que se vieron al principio del capítulo anterior. Tales fórmulas describen de manera precisa el rendimiento de los algoritmos que les corresponden: para obtener el tiempo de ejecución, se resuelven las recurrencias. Posteriormente se darán argumentos más rigurosos al estudiar algunos algoritmos concretos: por ahora lo que interesa son las fórmulas, no los algoritmos.

Fórmula 1. Esta recurrencia aparece en un programa recursivo que efectúa bucles en los datos de entrada para eliminar un elemento:

$$C_N = C_{N-1} + N, \quad \text{para } N \geq 2 \text{ con } C_1 = 1.$$

Solución: C_N es del orden de $N^2/2$. Para resolver esta recurrencia, se aplica sobre sí misma, «en cascada», de la siguiente forma:

$$\begin{aligned}
 N &= C_{N-1} + N \\
 &= C_{N-2} + (N-1)N \\
 &= C_{N-3} + (N-2) + (N-1) + N \\
 &\quad \cdot \\
 &\quad \cdot \\
 &= C_1 + 2 + \dots + (N-2) + (N-1) + N \\
 &= 1 + 2 + \dots + (N-2) + (N-1) + N \\
 &= \frac{N(N+1)}{2}.
 \end{aligned}$$

La evaluación de la suma $1 + 2 + \dots + (N-2) + (N-1) + N$ es elemental: el resultado obtenido anteriormente puede establecerse añadiendo la misma suma, pero en orden inverso, término a término. Este resultado, que es dos veces el valor buscado, contiene N términos, cada uno de los cuales vale $N+1$.

Fórmula 2. Esta recurrencia aparece en un programa recursivo que divide los datos de entrada en un solo paso:

$$C_N = C_{N/2} + 1, \quad \text{para } N \geq 2 \text{ con } C_1 = 0.$$

Solución: C_N es del orden de $\lg N$. Escrita de esta forma, esta ecuación carece de sentido a menos que N sea par o que se suponga que $N/2$ es una división entera: por ahora, se supone que $N = 2^n$, o lo que es lo mismo $n = \lg N$, de modo que la recurrencia siempre esté bien definida. Pero entonces la recurrencia en cascada llega incluso a ser aún más fácil que la anterior:

$$\begin{aligned}
 C_{2^n} &= C_{2^{n-1}} + 1 \\
 &= C_{2^{n-2}} + 1 + 1 \\
 &= C_{2^{n-3}} + 3 \\
 &\quad \cdot \\
 &\quad \cdot \\
 &= C_{2^0} + n \\
 &= n.
 \end{aligned}$$

Resulta que la solución exacta para cualquier N depende de las propiedades de la representación binaria de N , pero C_N es del orden de $\lg N$ para todo N .

Fórmula 3. Esta recurrencia aparece en un programa recursivo que divide los datos de entrada en dos, pero que debe examinar cada elemento de ellos.

$$C_N = C_{N/2} + N, \quad \text{para } N \geq 2 \text{ con } C_1 = 0.$$

Solución: C_N es del orden de $2N$. Reduciendo como antes se obtiene la suma $N + N/2 + N/4 + N/8 + \dots$ (como en el caso anterior, esta serie sólo tendrá sentido cuando N sea una potencia de dos). Si la sucesión fuera infinita, sería una serie geométrica simple que convergería a $2N$. Para cualquier valor de N , la solución exacta implica otra vez la representación binaria de N .

Fórmula 4. Esta recurrencia aparece en un programa recursivo que tiene que hacer un recorrido lineal de los datos de entrada, antes, durante o después de dividirla en dos partes:

$$C_N = 2C_{N/2} + N, \quad \text{para } N \geq 2 \text{ con } C_1 = 0.$$

Solución: C_N es del orden de $N\lg N$. Ésta es la solución que será la más citada, porque es el prototipo de muchos algoritmos del tipo de divide y vencerás.

$$\begin{aligned} C_{2^n} &= 2C_{2^{n-1}} + 2^n \\ \frac{C_{2^n}}{2^n} &= \frac{2C_{2^{n-1}}}{2^{n-1}} + 1 \\ &= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1 \\ &\quad \vdots \\ &= n. \end{aligned}$$

La solución se obtiene de la misma forma que la de la fórmula 2, pero con el truco adicional de dividir los dos miembros de la recurrencia por 2^n en el segundo paso, para hacer la recurrencia en cascada.

Fórmula 5. Esta recurrencia aparece en un programa recursivo que divide los datos de entrada en dos partes en un solo paso, como en el programa de dibujar una regla del Capítulo 5.

$$C_N = 2C_{N/2} + 1, \quad \text{para } N \geq 2 \text{ con } C_1 = 0.$$

Solución: C_N es del orden de $2N$. Esto se obtiene de la misma forma que en la fórmula 4.

Se pueden tratar variantes secundarias de estas fórmulas, con diferentes condiciones iniciales o ligeras diferencias en los términos añadidos, utilizando las mismas técnicas de resolución, aunque el lector debe tener en cuenta que algunas recurrencias que parecen similares a las anteriores, en realidad, pueden ser bastante difíciles de resolver. (Existe una variedad de técnicas generales avanzadas para tratar estas ecuaciones con rigor matemático.) Se encontrarán algunas recurrencias más complicadas en capítulos posteriores, dejándose el estudio de su solución hasta el momento en que aparezcan.

Perspectiva

Muchos de los algoritmos de este libro han sido objeto de análisis matemáticos detallados y de estudios de rendimiento demasiado complejos para tratarlos aquí. De hecho, basándose en tales estudios es posible recomendar la utilización de muchos de los algoritmos que se presentarán.

No todos los algoritmos se han sometido a análisis tan detallados; en efecto, durante el proceso de diseño es preferible trabajar con indicadores de rendimiento aproximados para poder guiar el proceso sin detalles extraños. Según se vaya refinando el diseño, se debe avanzar más en el análisis y se necesitará utilizar herramientas matemáticas más sofisticadas. A menudo, el proceso de diseño conduce a estudios detallados de la complejidad, que a su vez llevan a algoritmos «teóricos» muy alejados de cualquier aplicación particular. Es un error común suponer que el análisis aproximado de estudios de complejidad se traducirá inmediatamente en algoritmos prácticamente eficaces: esto suele conducir a sorpresas desagradables. Por otra parte, la complejidad de cálculo es una herramienta poderosa para obtener las condiciones de partida del diseño sobre las que pueden basarse nuevos métodos importantes.

No se debería utilizar un algoritmo sin tener una indicación de cómo llevarlo a cabo: las aproximaciones descritas en este capítulo ayudarán a proporcionar alguna indicación del rendimiento de una gran variedad de algoritmos, como los que se verán en los capítulos siguientes. En el próximo se presentarán otros factores importantes que influyen a la hora de elegir un algoritmo.

Ejercicios

1. Suponiendo que se sabe que el tiempo de ejecución de un algoritmo pertenece a $O(N \log N)$ y que el de otro pertenece a $O(N^3)$, ¿qué se puede decir sobre el rendimiento relativo de estos algoritmos?

2. Suponiendo que se sabe que el tiempo de ejecución de un algoritmo es siempre del orden de $N \log N$ y que el de otro pertenece a $O(N^3)$, ¿qué se puede decir sobre el rendimiento relativo de estos algoritmos?
3. Suponiendo que se sabe que el tiempo de ejecución de un algoritmo es siempre del orden de $N \log N$ y que el de otro es siempre del orden de N^3 , ¿qué se puede decir sobre el rendimiento relativo de estos algoritmos?
4. Explicar la diferencia entre $O(1)$ y $O(2)$.
5. Resolver la recurrencia

$$C_N = C_{N/2} + N^2, \quad \text{para } N \geq 2 \text{ con } C_1 = 0,$$

cuando N es una potencia de dos.

6. ¿Para qué valores de N se verifica $10N \lg N > 2N^2$?
7. Escribir un programa para calcular el valor exacto de C_N en la fórmula 2, como se presentó en el Capítulo 5. Comparar los resultados con $\lg N$.
8. Demostrar que la solución exacta de la fórmula 2 es $\lg N + O(1)$.
9. Escribir un programa recursivo para calcular el mayor de los enteros inferiores a $\log_2 N$. (Ayuda para $N > 1$, el valor de esta función para $N/2$ es una unidad mayor que N .)
10. Escribir un programa iterativo para resolver el problema del ejercicio anterior. Después escribir un programa que efectúe el cálculo utilizando subrutinas de la biblioteca de C++. Si la computadora lo permite, comparar el rendimiento de estos tres programas.

Implementación de algoritmos

Como se mencionó en el Capítulo 1, el principal objetivo de este libro son los algoritmos en sí mismos, de forma que, cuando se presente alguno de ellos, se tratará como si su rendimiento fuera el factor crucial para la realización correcta de tareas mayores. Este punto de vista se justifica porque estas situaciones aparecen con todos los algoritmos, y porque la búsqueda cuidadosa de soluciones eficaces para un problema conduce frecuentemente a otros algoritmos más elegantes (y más eficaces). Por supuesto, este planteamiento restrictivo es muy poco realista, ya que cuando se resuelve un problema complicado con una computadora deben tenerse en cuenta otros muchos factores. En este capítulo se presentarán cuestiones referentes a la forma de hacer útiles, en aplicaciones prácticas, los algoritmos algo idealizados que se describen en el libro.

Las propiedades del algoritmo, después de todo, son sólo una cara de la moneda, ya que una computadora puede utilizarse para resolver un problema de forma eficaz sólo si está suficientemente comprendido. El considerar cuidadosamente las propiedades de las aplicaciones está más allá del alcance de este libro. La intención es proporcionar suficiente información sobre los algoritmos básicos, de manera que cualquiera pueda tomar decisiones inteligentes sobre su empleo. La mayoría de los algoritmos que se tratarán aquí se utilizan en la práctica en muchas aplicaciones. La extensión de los algoritmos disponibles para resolver diferentes problemas depende de la extensión de las necesidades de las diversas aplicaciones. No existe «el mejor» algoritmo de búsqueda (por poner un ejemplo), pero un método puede ser idóneo en un sistema de reservas de unas líneas aéreas y otro podrá ser mejor para utilizarlo en el bucle interno de un programa de descifrado.

Los algoritmos rara vez existen en condiciones ideales, excepto posiblemente en la mente de sus diseñadores teóricos que inventan métodos sin pensar en ninguna implementación definitiva, o en la mente de los programadores de

sistemas de aplicaciones, que «amañan» métodos *ad hoc* para resolver problemas que por otra parte están bien delimitados. El diseño adecuado de un algoritmo supone el tener en cuenta el posible impacto que éste tendrá en las implementaciones posteriores, y la programación adecuada de las aplicaciones implica el tener en cuenta las características de rendimiento de los métodos básicos empleados.

Selección de un algoritmo

Como se verá en los capítulos siguientes, normalmente se dispondrá de varios algoritmos para resolver cada problema, todos con diferentes características de rendimiento, variando desde la simple solución de «fuerza bruta» (aunque probablemente ineficaz) hasta una solución compleja «bien afinada» (e incluso óptima). (En general, no es cierto que el algoritmo más eficiente sea el que tiene la implementación más complicada, ya que algunos de los mejores algoritmos son bastante elegantes y concisos. Pero para los fines de este estudio se supondrá que esta regla es cierta.) Como se argumentaba anteriormente, no se puede decidir qué algoritmo utilizar para un problema sin analizar las necesidades del mismo: ¿Con qué frecuencia se utilizará el programa?, ¿cuáles son las características generales del sistema de computación que se va a utilizar?, ¿es el algoritmo una parte pequeña de una gran aplicación, o viceversa?

La primera regla de la implementación es que se debe *implementar primero el algoritmo más simple que resuelva un problema dado*. Si el problema particular con el que se tropieza se resuelve fácilmente, entonces el algoritmo sencillo podría resolver el problema y no sería necesario hacer nada más; pero si se requiere un algoritmo más sofisticado, entonces la implementación sencilla proporciona una forma de comprobación para casos puntuales y una línea básica para evaluar las características del rendimiento.

Si sólo se va a ejecutar un algoritmo pocas veces, en casos que no son demasiado grandes, entonces seguramente es preferible que la computadora tarde un poco de más de tiempo en la ejecución de un algoritmo un poco menos eficaz, en lugar de que el programador emplee excesivo tiempo desarrollando una implementación sofisticada. Por supuesto, existe el peligro de que se pueda terminar utilizando el programa más de lo que se suponía originalmente, y por tanto conviene estar siempre preparado para volver a empezar e implementar un algoritmo mejor.

Si el algoritmo se va a integrar en un gran sistema, la implementación del método de la «fuerza bruta» proporciona la funcionalidad que se requiere de una manera fiable, y posteriormente podrá mejorarse el rendimiento (de una manera controlada), sustituyendo el algoritmo por otro más refinado. Por supuesto, cuando se estudie el comportamiento completo del sistema, se debería tener cuidado para no excluir opciones al implementar el algoritmo, de tal manera que sea difícil mejorarlo más adelante, y se debería tener un especial cui-

dado con aquellos algoritmos que dan lugar a cuellos de botella en la ejecución. En grandes sistemas, es frecuente que las especificaciones de diseño del sistema dicten desde el comienzo cuál es el mejor algoritmo. Por ejemplo, puede que una estructura de datos compartida por el sistema sea una lista enlazada o un árbol, por lo que son preferibles los algoritmos basados en estas estructuras particulares. Por otro lado, cuando se tomen decisiones a la escala del sistema se debe prestar mucha atención al elegir los algoritmos a utilizar porque al final es muy frecuente que el comportamiento de todo el sistema dependa de algún algoritmo básico, como los que se tratarán en este libro.

Si el algoritmo sólo se va a ejecutar unas cuantas veces, pero sobre problemas muy grandes, entonces se deseará asegurarse que se obtendrá una salida coherente, y tener una estimación de cuánto tiempo tardará. Aquí otra vez, una implementación sencilla puede ser a veces bastante útil para la resolución de una tarea larga, incluyendo el desarrollo de todo lo necesario para la comprobación de los resultados.

El error más común a la hora de seleccionar un algoritmo es ignorar las características de rendimiento. Los algoritmos más rápidos suelen ser los más complicados, y también es frecuente que las personas que los desarrollan estén dispuestas a aceptar un algoritmo más lento para evitar trabajar con una complejidad añadida. Pero, a menudo, un algoritmo más rápido no tiene por qué ser mucho más complicado, y trabajar con una pequeña complicación añadida es un pequeño precio a pagar para evitar manejar un algoritmo lento. Un número sorprendente de usuarios de sistemas de información pierden un tiempo considerable esperando que terminen de ejecutarse simples algoritmos cuadráticos, cuando existen algoritmos cuya complejidad en tiempo está próximo a $N \log N$, y que pueden ejecutarse en una fracción de dicho tiempo.

El segundo error más común que se comete, a la hora de seleccionar un algoritmo, es dar excesiva importancia a las características de rendimiento. Un algoritmo en $N \log N$ podría ser ligeramente más complicado que un algoritmo cuadrático para resolver el mismo problema; pero un algoritmo en $N \log N$, más eficaz, podría dar lugar a un incremento sustancial de la complejidad (y en realidad podría ser más rápido sólo para valores de N muy grandes). También ocurre que muchos programas de hecho sólo se ejecutan unas pocas veces: el tiempo necesario para implementar y depurar un algoritmo optimizado podría ser considerablemente mayor que el tiempo que se necesita para ejecutar uno sencillo que es un tanto más lento.

Análisis empírico

Como se mencionó en el Capítulo 6, por desgracia frecuentemente se da el caso de que el análisis matemático aporte muy poca luz sobre cómo es el comportamiento esperado de un algoritmo particular en una situación concreta. En tales casos, es necesario realizar un *análisis empírico*, en el que se implementa

cuidadosamente un algoritmo y se controla su ejecución con una entrada «típica». De hecho, esto debería realizarse incluso cuando se disponga de los resultados matemáticos completos, con el fin de comprobar su validez.

Dados dos algoritmos que resuelvan el mismo problema, el método es muy claro: ¡se ejecutan los dos para ver cuál de ellos lleva más tiempo! Decir esto podría parecer demasiado obvio, pero probablemente sea la omisión más común en el estudio comparativo de algoritmos. El hecho de que un algoritmo sea diez veces más rápido que otro es muy improbable que se le escape a alguien que espera que uno de ellos acabe en tres segundos y que el otro acabe en treinta, pero es muy fácil que se le pase por alto algo como un pequeño factor constante en un análisis matemático.

Sin embargo, también es fácil cometer errores cuando se comparan implementaciones, en especial si intervienen diferentes máquinas, compiladores o sistemas, o si están comparando programas muy grandes con entradas mal especificadas. Desde luego, uno de los factores que condujo al desarrollo del análisis matemático de los algoritmos fue la tendencia a confiar en los «modelos estándar» cuyo comportamiento probablemente se entienda mejor a través de un análisis cuidadoso.

El principal peligro que se corre al comparar empíricamente programas es que una implementación puede ser más «optimizada» que otra. Es probable que el inventor de un nuevo algoritmo preste una atención muy cuidadosa a cada aspecto de su implementación, pero no a los detalles de la implementación del algoritmo clásico rival del suyo. Para confiar en la precisión de un estudio de comparación empírico hay que estar seguro de que se ha prestado la misma atención a ambas implementaciones. Afortunadamente, el caso más frecuente es el siguiente: muchos algoritmos excelentes se han obtenido haciendo modificaciones relativamente pequeñas de otros algoritmos creados para resolver el mismo problema, siendo válidos los estudios comparativos.

Un caso particular importante surge cuando se compara un algoritmo con otra versión de *sí mismo* o cuando se comparan implementaciones ligeramente diferentes. Una buena manera de comprobar la eficacia de una modificación en particular, o de otra idea de la misma implementación, es ejecutar ambas versiones con alguna entrada «tipo» y en consecuencia seleccionar la más rápida. De nuevo, parece obvio mencionarlo, pero ¡el usuario debe tener cuidado!, porque hay un sorprendente número de investigadores dedicados al diseño de algoritmos que nunca implementan sus diseños.

Como antes se esbozó, y también al comienzo del Capítulo 6, el criterio adoptado aquí es que diseño, implementación, análisis matemático y análisis empírico (todos ellos conjuntamente) contribuyen de forma crucial al desarrollo de unas buenas implementaciones de algoritmos. Se utilizan todas las herramientas disponibles para obtener toda la información sobre las propiedades de los programas, y después se los modifica o se desarrollan programas nuevos, a partir de dicha información. Por otro lado, no siempre está justificado hacer un gran número de cambios pequeños con la esperanza de mejorar ligeramente la ejecución. A continuación se tratará esta cuestión con más detalle.

Optimización de un programa

El procedimiento general para modificar un programa, con la finalidad de conseguir otra versión más rápida en su ejecución, se denomina *optimización del programa*. Éste no es el término adecuado, porque es poco probable encontrar una implementación que sea «la mejor», pero, aunque no se pueda optimizar el programa, se puede esperar mejorarlo. Normalmente, la optimización del programa se realiza de forma automática, como parte del proceso de compilación, para mejorar el rendimiento del código compilado. Aquí se utiliza el término para referirse a las mejoras *específicas del algoritmo*. Por supuesto, el proceso también depende bastante del entorno de programación y de la máquina utilizada; por tanto aquí sólo se consideran cuestiones generales y no técnicas específicas.

Este tipo de actividad está justificada sólo si se está seguro de que el programa se empleará muchas veces o sobre grandes conjuntos de datos y si la experimentación demuestra que el esfuerzo dedicado a mejorar la implementación será recompensado con una mejor ejecución. La mejor forma de perfeccionar el rendimiento de un algoritmo es mediante un proceso gradual de transformación en mejores programas y mejores implementaciones. La eliminación de la recursión del Capítulo 5 es un ejemplo de un proceso de este tipo, aunque la forma de mejorar el rendimiento no había sido el objetivo en ese momento.

El primer paso en la implementación de un algoritmo es desarrollar una versión inicial en su forma más simple. Esto proporciona una línea básica para posteriores refinamientos y mejoras y, como se mencionó anteriormente, es muy frecuente que haya que realizar todo esto. Se deben contrastar los resultados matemáticos disponibles con los resultados de la implementación; por ejemplo, si el análisis indica que el tiempo de ejecución es $O(\log N)$, pero el tiempo de ejecución real es de varios segundos, entonces algo falla, o bien la implementación, o bien el análisis, y ambos deben estudiarse con más cuidado.

El siguiente paso es identificar el «bucle interno» y tratar de minimizar el número de instrucciones que lo componen. Quizás la manera más fácil de encontrar este bucle sea ejecutar el programa y comprobar qué instrucciones se ejecutan más a menudo. Por lo regular, esto da una buena indicación de dónde se puede mejorar el programa. Cada instrucción del bucle interno debería someterse a un cuidadoso examen: ¿Es realmente necesaria?, ¿existe una manera más eficaz de llevar a cabo la misma tarea? Por ejemplo, por lo general merece la pena codificar algo más, para eliminar llamadas a procedimiento desde el bucle interno. Existen otras técnicas «automáticas» para hacer esto, muchas de las cuales están implementadas en compiladores estándar. A final de cuentas, el mejor rendimiento se logra traduciendo el bucle interno a lenguaje de máquina o lenguaje ensamblador; pero esto suele ser un último recurso.

En realidad no todas las «mejoras» producen ganancias en el rendimiento, de modo que es sumamente importante comprobar el alcance del ahorro obte-

nido en cada paso. Además, a medida que la implementación se perfecciona más y más, es aconsejable volver a comprobar si está justificada esta profundización en los detalles del código. En el pasado, el tiempo de cálculo era tan costoso que estaba casi siempre justificado emplear más tiempo de programación para ahorrar ciclos de cálculo, pero las cosas han cambiado en los últimos años.

Por ejemplo, considerando el algoritmo del recorrido del árbol en orden previo presentado en el Capítulo 5, la eliminación de la recursión es en realidad el primer paso para «optimizar» este algoritmo, porque su acción se centra en el bucle interno. La versión no recursiva dada es probablemente más lenta que la versión recursiva en muchos sistemas (puede comprobarlo el lector) porque el bucle interno es más grande e incluye cuatro llamadas (aunque no recursivas) a procedimientos (sacar, meter, meter y vacía), en vez de dos. Si se reemplazan las llamadas a los procedimientos de la pila por otro código, para acceder directamente a la pila (utilizando, por ejemplo, una implementación por array), es probable que este programa será significativamente más rápido que la versión recursiva. (Una de las operaciones push del algoritmo es provisional, por lo que el programa estándar de un bucle dentro de otro constituye probablemente la base de la versión optimizada.) Es evidente que el bucle interno implica incrementar el puntero de la pila, almacenar un puntero ($t \rightarrow \text{der}$) en el array de la pila, reinicializar el puntero t (a $t \rightarrow \text{izq}$) y compararlo con z . En muchas máquinas, esto se podría implementar con cuatro instrucciones de lenguaje de máquina, mientras que un compilador típico es probable que genere el doble o más. Es posible hacer sin demasiado trabajo que el programa se ejecute cuatro o cinco veces más rápido que la implementación recursiva directa.

Obviamente, los problemas que se están estudiando aquí dependen en gran medida del sistema y de la máquina. No es conveniente embarcarse en un intento serio de acelerar un programa, sin tener un conocimiento bastante detallado del sistema operativo y del entorno de programación. La versión óptima de un programa se puede volver bastante frágil y difícil de modificar, y un compilador o un sistema operativo nuevos (por no mencionar una computadora nueva) podría arruinar por completo una implementación cuidadosamente optimizada. Se debe tener una precaución especial cuando se utiliza un lenguaje evolucionado como el C++, en el que hay que esperar frecuentes modificaciones y mejoras.

Por otra parte, el libro se enfoca a la eficacia de las implementaciones prestando especial atención al bucle interno y asegurándose de que el tiempo de ejecución del algoritmo está minimizado en su mayor parte. Los programas están codificados de forma escueta y flexible con el fin de poder añadir algunas mejoras, de una manera directa y en cualquier entorno de programación concreto.

La implementación de un algoritmo es un proceso cíclico del desarrollo de un programa: se concibe, se depura, se estudian sus características y después se refina la implementación hasta que se alcanza el nivel de rendimiento deseado. Como se vio en el Capítulo 6, en general, el análisis matemático puede ayudar en el proceso: primero, para sugerir qué algoritmos son susceptibles de

llevar a cabo una cuidadosa implementación; segundo, para ayudar a comprobar que la implementación se desarrolla como se esperaba. En algunos casos, este proceso puede conducir al descubrimiento de ciertas propiedades, que hacen posible un nuevo algoritmo o mejoras sustanciales de una versión más antigua.

Algoritmos y sistemas

Las implementaciones de los algoritmos de este libro se pueden encontrar en una amplia variedad de programas, sistemas operativos y sistemas de aplicaciones. La intención es describir los algoritmos y animar al lector a que centre su atención en sus propiedades dinámicas experimentadas con las implementaciones dadas. En algunas aplicaciones, las implementaciones pueden ser útiles tal y como vienen dadas, pero para otras puede necesitarse más trabajo.

Como se mencionó en el Capítulo 2, los programas de este libro sólo utilizan aspectos básicos del C++, sin aprovechar las posibilidades más potentes, disponibles en C++ y en otros entornos de programación. La finalidad es el estudio de los algoritmos, no la programación de sistemas o aspectos avanzados de los lenguajes de programación. Se supone que los aspectos esenciales de los algoritmos se exponen mejor a través de implementaciones sencillas y directas en un lenguaje casi universal.

El estilo de programación que se utiliza es conciso, con nombres de variables cortos y con pocos comentarios, de manera que se destaquen las estructuras de control. La «documentación» de los algoritmos es el texto que los acompaña. Se espera que los lectores que utilicen estos programas en aplicaciones reales los desarrollem adaptándolos para su uso particular. Al construir sistemas reales está justificado un estilo de programación más «defensivo»: los programas deben implementarse de modo que puedan modificarse fácilmente, leerse con rapidez y entenderse por otros programadores, y que además realicen una buena interfaz con otras partes del sistema.

En particular, aunque los algoritmos que se tratan sean apropiados para estructuras de datos más complejas, las estructuras de datos que se necesitan para las aplicaciones normalmente contienen bastante más información de la que se utiliza en el libro. Por ejemplo, se habla de búsqueda en archivos que contienen números enteros o cadenas de caracteres cortas, mientras que, por lo regular, una aplicación necesita operar sobre largas cadenas de caracteres que forman parte de grandes registros. Pero los métodos básicos disponibles en ambos casos son los mismos. En tales casos se tratarán aspectos destacados de cada algoritmo y se verá cómo se podrían relacionar con diversas características o necesidades de la aplicación.

Muchos de los comentarios anteriores tienen que ver con la mejora del rendimiento de un algoritmo particular y también se aplican para mejorar el rendimiento de un sistema grande. Sin embargo, a gran escala, una de las técnicas

para mejorar el rendimiento de un sistema podría ser reemplazar un módulo en el que se implementa un algoritmo por un módulo en el que se implementa otro. Un principio básico para construir grandes sistemas es que deberían ser posibles tales cambios. Normalmente, cuando un sistema evoluciona se obtienen conocimientos más precisos sobre las necesidades específicas de los módulos en particular. Este conocimiento específico hace posible una selección más cuidadosa del mejor algoritmo a utilizar entre los que satisfacen esas necesidades; después se puede concentrar el esfuerzo en mejorar el rendimiento de ese algoritmo, como se dijo anteriormente. Es cierto que la mayor parte del código del sistema se ejecuta sólo unas pocas veces (o ninguna) y que el principal interés del constructor del sistema es crear un todo coherente. Por otra parte, también es muy probable que, cuando el sistema entre en funcionamiento, muchos de sus recursos se dedicarán a resolver problemas fundamentales del mismo tipo que los presentados en este libro, de modo que es conveniente que el constructor del sistema conozca los algoritmos básicos que aquí se describen.

Ejercicios

1. ¿Cuánto tiempo se tarda en contar hasta 100.000? Dar una estimación del tiempo que tardaría el programa `j = 0; for (i = 1; i < 100000; i++) j++;` en el entorno de programación del lector. Después ejecutar el programa para comprobar dicha estimación.
2. Responder a la pregunta anterior utilizando `repeat` y `while`.
3. Ejecutándolo para valores pequeños, estimar el tiempo que llevaría la implementación de la criba de Eratóstenes del Capítulo 3 para un valor de $N = 1.000.000$ (si se dispone de memoria suficiente).
4. «Optimizar» la implementación de la criba de Eratóstenes del Capítulo 3 para encontrar el número primo más grande que se pueda obtener en 10 segundos de cálculo.
5. Demostrar la afirmación del texto según la cual, al eliminar la recursión del algoritmo de recorrido del árbol en orden previo del Capítulo 5 (con llamadas a procedimientos para operaciones con pilas), el programa se hace más lento.
6. Demostrar la afirmación del texto según la cual, al eliminar la recursión del algoritmo de recorrido del árbol en orden previo del Capítulo 5 (e implementando directamente operaciones de pila), el programa se hace más rápido.
7. Examinar el programa en lenguaje ensamblador producido por el compilador C++ del entorno local de programación del lector para el algoritmo recursivo de recorrido de árbol en orden previo del Capítulo 5.
8. Diseñar un experimento para comprobar cuál de las dos implementaciones de una pila, lista enlazada o array es más eficaz en el entorno de programación del lector.

9. Determinar cuál es el método más eficaz para representar la regla dada en el Capítulo 5: ¿el método recursivo o el no recursivo?
10. En la implementación no recursiva dada en el Capítulo 5, al recorrer un árbol completo de $2^n - 1$ nodos en orden previo, ¿cuántas inserciones se utilizan exactamente en la pila?

REFERENCIAS para Fundamentos

Existe un gran número de libros de texto de introducción a la programación y a las estructuras de datos elementales. La referencia estándar para C++ es el libro de Stroustrup, y la mejor fuente para cuestiones específicas y ejemplos de programas en C, con el mismo espíritu que el encontrado en este libro, es el libro de Kernighan y Ritchie sobre este lenguaje. La colección más completa de información sobre las propiedades de estructuras de datos elementales y árboles es el Volumen 1 de Knuth: los Capítulos 3 y 4 no contemplan más que una pequeña parte de la información que se proporciona en el libro de Knuth.

La referencia clásica para el análisis de algoritmos basado en las medidas del comportamiento asintótico del peor caso es el libro de Aho, Hopcroft y Ullman. Los libros de Knuth cubren, con mayor amplitud, el análisis del caso medio y son la fuente autorizada sobre las propiedades específicas de varios algoritmos (por ejemplo, casi 50 páginas del Volumen 2 se dedican al algoritmo de Euclides.) El libro de Gonnet trata tanto el análisis del peor caso como el del caso medio, así como muchos algoritmos de reciente desarrollo.

El libro de Graham, Knuth y Patashnik comprende los aspectos matemáticos que normalmente se utilizan en el análisis de algoritmos. Por ejemplo, dicho libro describe muchas técnicas para resolver ecuaciones de recurrencia como las dadas en el Capítulo 6 y otras mucho más difíciles que se encontrarán más adelante. Tal materia también se trata con gran amplitud en los libros de Knuth.

El libro de Roberts abarca la materia relativa al Capítulo 6, y los libros de Bentley plantean el mismo punto de vista que el Capítulo 7 y secciones posteriores de este libro. Bentley describe de forma detallada un gran número de estudios completos de casos sobre la evaluación de varias propuestas para desarrollar algoritmos e implementaciones para resolver algunos problemas interesantes.

- A. V. Aho, J. E. Hopcroft y J. D. Ullman, *The Design and Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1975.
- J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1985; *More Programming Pearls*, Addison-Wesley, Reading, MA, 1988.
- G. H. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1984.
- R. L. Graham, D. E. Knuth y O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1988.
- B. W. Kernighan y D. M. Ritchie, *The C Programming Language*, segunda edición, Prentice Hall, Englewood Cliffs, NJ, 1988.
- D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, segunda edición, Addison-Wesley, Reading, MA, 1973; *Volume 2: Seminumerical Algorithms*, segunda edición, Addison-Wesley, Reading, MA, 1981; *Volume 3: Sorting and Searching*, segunda impresión, Addison-Wesley, Reading, MA, 1975.

- E. Roberts, *Thinking Recursively*, John Wiley & Sons, Nueva York, 1986.
- B. Stroustrup, *The C++ Programming Language*, segunda edición, Addison-Wesley, Reading, MA, 1991. (Existe versión en español por Addison-Wesley Iberoamericana y Ediciones Díaz de Santos, *N. del E.*)

Algoritmos de ordenación

Métodos de ordenación elementales

Durante esta primera excursión por el área de los algoritmos de ordenación, se verán algunos métodos «elementales» que son apropiados para archivos pequeños o con una estructura particular. Existen varias razones para estudiar con detalle estos algoritmos de ordenación sencillos: la primera es que proporcionan una forma relativamente fácil de aprender la terminología y los mecanismos básicos de los algoritmos de ordenación, con el fin de obtener una información previa adecuada para el estudio de los algoritmos más sofisticados. La segunda razón es que hay un gran número de aplicaciones de ordenación en las que es mejor utilizar estos métodos sencillos en lugar de otros más potentes pero con fines más generales. Por último, algunos de los métodos sencillos se pueden extender a otros métodos de carácter general o pueden utilizarse para mejorar la eficacia de métodos más potentes.

Como se acaba de mencionar, existen varias aplicaciones de ordenación en las que el método elegido puede ser un algoritmo relativamente sencillo. Con frecuencia, los programas de ordenación se usan una única vez (o muy pocas veces). Si el número de elementos a ordenar no es demasiado grande (por ejemplo, menos de 500), puede ser más eficaz utilizar un método sencillo en lugar de implementar y depurar uno complicado. Los métodos elementales siempre son apropiados para archivos pequeños (menos de 50 elementos), y es poco probable que se pueda justificar el uso de un algoritmo sofisticado para ordenar un archivo pequeño, salvo que vaya a clasificarse un gran número de archivos. Otros tipos de archivos que son relativamente fáciles de ordenar son aquellos que ya están casi ordenados (o ya están ordenados), o aquellos que contienen un gran número de claves iguales. Para estos archivos «bien estructurados» puede resultar mucho mejor utilizar métodos sencillos en lugar de métodos más generales.

Por regla general, los métodos elementales que se verán en el libro necesitan aproximadamente N^2 pasos para ordenar N elementos organizados al azar.

Cuando N es suficientemente pequeño esto no presenta ningún problema, y, si los elementos no están organizados aleatoriamente, alguno de estos métodos puede resultar mucho mejor que otros más sofisticados. Sin embargo, debe hacerse hincapié en que estos métodos *no* deberían utilizarse para ordenar archivos grandes, ni para ordenar archivos clasificados aleatoriamente, con la notable excepción de la ordenación de Shell, que es, en realidad, el método de ordenación elegido para muchas aplicaciones.

Reglas del juego

Antes de considerar algunos algoritmos específicos resultará útil presentar cierta terminología general y algunos supuestos básicos de los algoritmos de ordenación. Se hará el estudio de métodos para ordenar *archivos de registros* que contienen *claves*, que no son más que una parte de los registros (a menudo una pequeña parte), pero que se utilizan para controlar la ordenación. El objetivo de un método de ordenación es volver a organizar los registros para que sus claves estén ordenadas de acuerdo con alguna regla bien definida (por lo regular, en orden numérico o alfabetico).

Si el archivo a ordenar se encuentra en la memoria (o, en el contexto del libro, en un array de C++), entonces el método de ordenación se denomina *interno*. Cuando se realiza la ordenación de archivos situados en cinta o en disco se denomina ordenación *externa*. La diferencia principal entre las dos es que en una ordenación interna se puede acceder fácilmente a cualquier registro, mientras que en una externa debe accederse a los registros de un modo secuencial, o al menos en grandes bloques. En el Capítulo 13 se verán algunas ordenaciones externas, pero la mayoría de los algoritmos que se tratarán en el libro serán ordenaciones internas.

Como de costumbre, el principal parámetro de rendimiento que interesará es el tiempo de ejecución de los algoritmos de ordenación. El primero de los cuatro métodos que se presentan en este capítulo necesita un tiempo proporcional a N^2 , siendo N el número de elementos a ordenar, mientras que otros métodos más avanzados pueden ordenar N elementos en un tiempo proporcional a $M \log N$. (Puede demostrarse que ningún algoritmo de ordenación puede utilizar menos de $M \log N$ comparaciones entre claves.) Después de examinar estos métodos sencillos se estudiarán otros más avanzados que pueden ejecutarse en un tiempo proporcional o menor a $N^{3/2}$ y se verá que existen métodos que utilizan la propiedades digitales de las claves para obtener un tiempo de ejecución total proporcional a N .

El segundo factor importante a considerar es la cantidad de memoria extra necesaria para cada algoritmo de ordenación. Básicamente, se distinguirán tres tipos de métodos: aquellos que ordenan *in situ* y no utilizan memoria extra, salvo una eventual pila o tabla de pequeño tamaño; aquellos que utilizan una representación por lista enlazada y necesitan por tanto N palabras de memoria suplementaria.

mentaria para los punteros de la lista, y aquellos que necesitan bastante memoria extra para almacenar una copia del array que se desea ordenar.

Una característica de los métodos de ordenación, que a veces resulta importante en la práctica, es la *estabilidad*. Se dice que un método de ordenación es *estable* si, cuando se encuentra con dos registros que tienen la misma clave, conserva su orden relativo en el archivo. Por ejemplo, si se toma una lista con los estudiantes de una clase ordenados alfabéticamente y se ordena por notas, un método estable dará una lista en la que los estudiantes que tienen las mismas notas permanecen ordenados alfabéticamente; en cambio, un método inestable es probable que dé una lista sin que quede ningún rastro del orden alfabético original. La mayoría de los métodos sencillos son estables, pero la mayor parte de los algoritmos sofisticados conocidos no lo son. Si la estabilidad es vital, es posible imponerla añadiendo un pequeño índice a cada clave antes de la ordenación, o prolongando el alcance de la clave de alguna otra forma. La estabilidad parece que se adquiere fácilmente y a menudo se reacciona con desconfianza ante los efectos desagradables de la inestabilidad. De hecho, pocos métodos logran estabilidad sin utilizar grandes cantidades de espacio o tiempo suplementarios.

El siguiente programa tiene por objeto ilustrar los convenios generales que se utilizarán en este capítulo. Está formado por un programa principal que lee N números y a continuación llama a una subrutina para ordenarlos. En este ejemplo, solamente se ordenan los tres primeros números leídos: lo importante es que este programa «piloto» podría llamar a cualquier programa de ordenación en lugar de ordenar3.

```
inline void intercambio(tipoElemento a[], int i, int j)
{ tipoElemento t = a[i]; a[i] = a[j]; a[j] = t; }
ordenar3(tipoElemento a[],int N)
{
    if (a[1] > a[2]) intercambio(a, 1, 2);
    if (a[1] > a[3]) intercambio(a, 1, 3);
    if (a[2] > a[3]) intercambio(a, 2, 3);
}
const int maxN = 100;
main()
{
    int N, i; tipoElemento v, a[maxN+1];
    N = 0; while (cin >> v) a[++N] = v;
    a[0] = 0;
    ordenar3(a,N);
    for (i = 1; i<= N; i++) cout << a[i] << ' ';
    cout << '\n';
}
```

Como en el Capítulo 3, se mantiene la atención en los detalles de los algoritmos dejando sin especificar el tipo de los elementos a ordenar (`tipoElemento`), y se emplean aquellos algoritmos que permiten ordenaciones sencillas de arrays de números enteros en orden numérico, o de caracteres en orden alfabetico. En C++ es fácil utilizar `typedef` o plantillas para adaptar tales algoritmos de forma que se puedan utilizar en aplicaciones prácticas que puedan implicar grandes claves o registros.

Por lo regular, los programas de ordenación acceden a los registros de una de estas dos formas: o acceden a las claves para compararlas o acceden a los registros completos para intercambiarlos. La mayoría de los algoritmos que se estudiarán pueden describirse por medio de estas dos operaciones sobre registros arbitrarios. Si se van a ordenar registros grandes, será prudente hacer una «ordenación indirecta» para evitar desplazarlos durante el proceso: no se reorganizan los propios registros sino un array de punteros (o índices), de forma que el primer puntero apunte al registro más pequeño, etc. Las claves se pueden guardar ya sea con los registros (si son grandes) o bien con los punteros (si son pequeñas). Si es necesario, se pueden reorganizar los registros después de la ordenación, como se describirá más adelante en este capítulo.

El procedimiento `intercambio` lleva a cabo una operación de «intercambio» y es `inline` porque los intercambios son fundamentales para muchos programas de ordenación y normalmente forman parte del bucle interno. En realidad, el programa utiliza un acceso al archivo todavía más restringido: hay tres instrucciones de la forma «comparar dos registros y, si es necesario, intercambiarlos poniendo en primer lugar el de clave más pequeña». Los programas que se reducen a estas instrucciones son interesantes porque favorecen su implementación en cualquier máquina. Este punto se estudiará con más detalle en el Capítulo 40.

Mientras se pueda dar alguna indicación de cómo explotar las facilidades que ofrece C++ para construir diversos algoritmos que se consideran útiles para ciertas aplicaciones, se evitará insistir en el problema general de cómo se deberían «empaquetar» las ordenaciones. Por ejemplo, ya se ha tocado el tema de la utilización de plantillas o de `typedef` para hacer que los algoritmos puedan ser útiles para más tipos de claves. Otro ejemplo: es razonable pasar el array a ordenar como un parámetro de la rutina de ordenación en C++, pero esto no tiene por qué ser así en otros lenguajes de programación. En lugar de ello, ¿debería trabajar el programa sobre un array global?, ¿debería la rutina de ordenación ser parte de una clase que generalice las operaciones que se pueden llevar a cabo a cualquier array susceptible de ordenación? En algunos sistemas operativos es bastante fácil juntar programas sencillos, como por ejemplo el anterior, para que sirvan de «filtros» entre su entrada y su salida. Al contrario, muchas aplicaciones no necesitan realmente mecanismos tales como clases y filtros, siendo preferible introducir en la aplicación un pequeño código de ordenación. Claro está, estos comentarios se pueden aplicar a otros muchos algoritmos que se examinarán en este libro, pero el estudio de los algoritmos de ordenación descubrirá gran parte de los puntos más interesantes.

Tampoco se incluyen en los programas muchas instrucciones de «verificación de errores», aunque suele ser prudente hacerlo en las aplicaciones. Por ejemplo, la rutina piloto debería comprobar que N no tome un valor superior a $\max N$ (y ordenar3 debería verificar que $N=3$). Otra comprobación útil sería que el programa piloto se asegurara de que el array está ordenado después de llamar a ordenar3. Esto no garantiza que el programa de ordenación funcione (¿por qué?), pero puede ayudar a mostrar los errores.

Algunos programas utilizan otras variables globales, en cuyo caso las declaraciones que no son obvias se incluirán en el código del programa. Además, a menudo se reservará $a[0]$ (y algunas veces $a[N+1]$) para almacenar las claves especiales utilizadas por algunos de los algoritmos. En los ejemplos se utilizarán con frecuencia las letras del alfabeto en lugar de los números: aquéllas se pueden emplear de manera evidente utilizando las funciones estándar de C++ que convierten enteros a caracteres, y viceversa.

Ordenación por selección

Uno de los algoritmos de ordenación más sencillos funciona de la siguiente forma: primero se busca el elemento más pequeño del array y se intercambia con el que está en la primera posición; después se busca el segundo elemento más pequeño y se intercambia con el que está en la segunda posición, continuándose de esta forma hasta que todo el array esté ordenado. Este método se denomina *ordenación por selección* porque funciona «seleccionando» repetitivamente el elemento más pequeño de los que quedan por ordenar, como muestra la Figura 8.1. En el primer paso, la A de la octava posición es el elemento más pequeño, por lo que se cambia por la E del principio. En el segundo paso, la segunda A es el elemento más pequeño de los que quedan, de forma que se intercambia con la J de la segunda posición. Después la D se intercambia con la E de la tercera posición, y a continuación, en el cuarto paso, la primera E se intercambia con la M de la cuarta posición, y así sucesivamente.

El siguiente programa es una implementación de este proceso. Para todo i entre 1 y $N-1$, se intercambia $a[i]$ con el elemento más pequeño de la sucesión $a[i], \dots, a[N]$:

```
void seleccion (tipoElemento a[], int N)
{
    int i, j, min;
    for (i = 1; i > N; i++)
    {
        min = i;
```

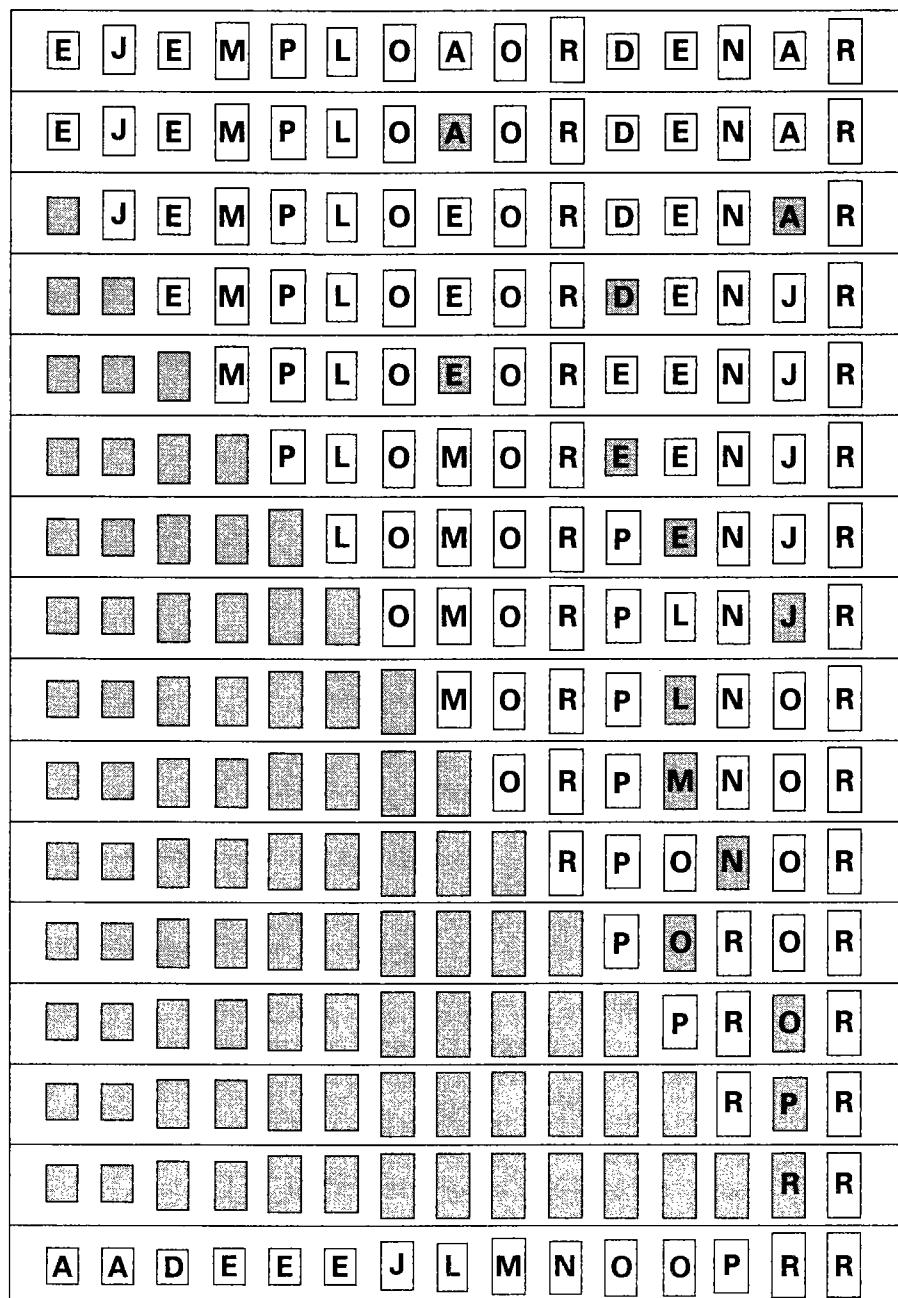


Figura 8.1 Ordenación por selección.

```

        for (j = i+1; j <= N; j++)
            if (a[j] < a[min]) min = j;
        intercambio (a, min, i);
    }
}

```

A medida que el índice *i* recorre el archivo de izquierda a derecha, los elementos que quedan a su izquierda están ya en su posición definitiva dentro del array (y no se desplazarán otra vez), de manera que el array está completamente ordenado cuando el índice llega al extremo de la derecha.

Éste es uno de los métodos de ordenación más sencillos y funcionará muy bien con archivos pequeños. El «bucle interno» está formado por la comparación $a[j] < a[min]$ (más el código necesario para incrementar *j* y comprobar que no es mayor que *N*), y prácticamente no puede ser más simple. Más adelante se examinará el número de veces que es probable que se ejecuten estas instrucciones.

Además, a pesar de su enfoque evidente de «fuerza bruta», la ordenación por selección tiene de hecho una aplicación bastante importante: como la mayoría de los elementos se mueven como máximo una vez, este tipo de ordenación es el método que debe elegirse para ordenar archivos que tienen registros muy grandes y claves muy pequeñas. Esto se verá con detalle más adelante.

Ordenación por inserción

La *ordenación por inserción* es un algoritmo casi tan sencillo como la ordenación por selección, pero probablemente más flexible. Es el método que se utiliza a menudo para ordenar las cartas cuando se juegan unas manos de bridge: consideréngase los elementos uno tras otro, insertando cada uno en su lugar apropiado entre los que ya se han considerado (manteniéndolos ordenados). Como se muestra en la Figura 8.2, el elemento considerado se inserta simplemente moviendo una posición a la derecha a todos los elementos mayores que él e insertando a continuación el elemento en la posición vacante. La J de la segunda posición es mayor que la E de la primera, por lo que no hay que desplazarla. Al encontrar la E en la tercera posición se cambia con la J para poner E E J en el orden deseado, y así sucesivamente.

Este proceso se implementa en el programa siguiente. Para cada *i*, con valores entre 2 y *N*, se ordenan los elementos $a[1], \dots, a[i]$ insertando $a[i]$ en su lugar en la lista ordenada de elementos $a[1], \dots, a[i-1]$:

```

void insercion(tipoElemento a[], int N)
{
    int i, j; tipoElemento v;

```

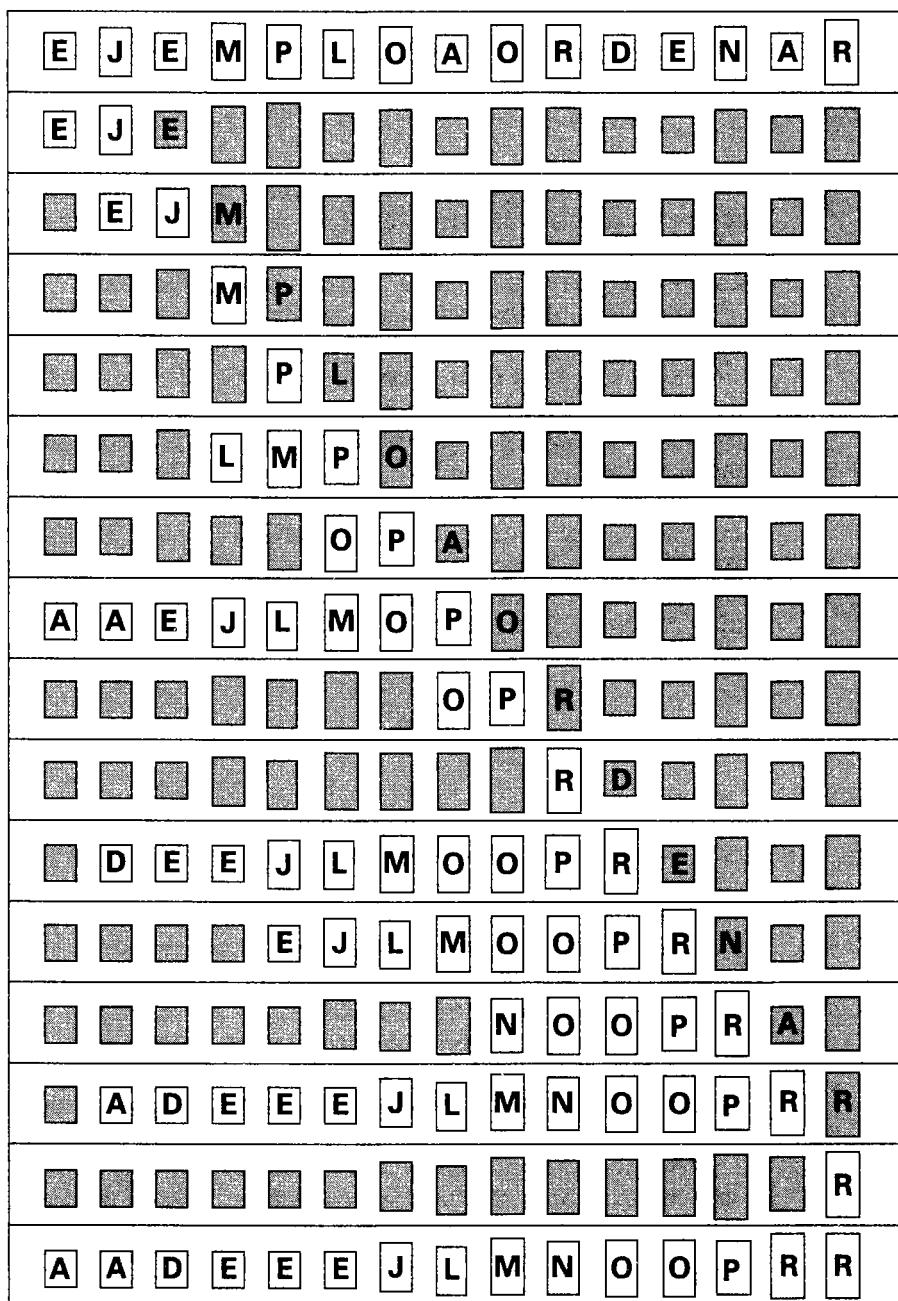


Figura 8.2 Ordenación por inserción.

```

for (i = 2; i <= N; i++)
{
    while (a[j-1] > v)
        { a[j] = a[j-1]; j--; }
    a[j] = v;
}

```

Como en una ordenación por selección, los elementos situados a la izquierda del índice i están ordenados entre sí durante la ordenación, pero no están en su posición definitiva, ya que puede ocurrir que tengan que moverse para hacer sitio a elementos más pequeños que se encuentren posteriormente. Sin embargo, el array está ordenado por completo cuando el índice alcanza el extremo derecho.

Hay que considerar otro detalle más importante: ¡el procedimiento de inserción no trabaja con la mayor parte de los datos de entrada! Esto es así porque cuando v sea el elemento más pequeño del array, el bucle `while` seguirá funcionando cuando se llegue al final del array por la izquierda. Para arreglar esto, se coloca una clave «centinela» en $a[0]$, haciendo que tome un valor inferior o igual al del elemento más pequeño del array. Los centinelas normalmente se utilizan en situaciones como éstas para evitar tener que incluir una comprobación (en este caso sería comprobar que $j > 1$), que casi siempre se produce en el bucle interno.

Si por alguna razón no es conveniente utilizar un centinela (por ejemplo, si no se puede definir fácilmente cuál es la clave más pequeña), entonces se podría utilizar la comprobación `while j > 1 && a[j-1] > v`. Esta solución no es atractiva, porque el caso de $j=1$ sólo se da en raras ocasiones, de manera que ¿por qué se comprueba con tanta frecuencia esta situación en el bucle interno? Es de destacar que cuando j es igual a 1, la comprobación anterior no accederá a $a[j-1]$ porque ésta es la forma como se evalúan las expresiones lógicas en C++ —en estos casos otros lenguajes podrían hacer accesos ilegales al array—. Otra forma de tratar esta situación en C++ es utilizar un `break` o un `goto` a la salida del bucle `while`. (Algunos programadores prefieren evitar las instrucciones `goto` y para ello son capaces de cualquier cosa, como por ejemplo llevar a cabo una acción dentro del bucle para asegurarse de que éste termina bien. En este caso, esta solución no parece que esté justificada, ya que no clarifica el programa y añade sobrecargas cada vez que se recorre el bucle como protección contra un caso raro.)

Digresión: Ordenación de burbuja

Un método de ordenación elemental que se enseña a menudo en los cursos de introducción a la informática es la *ordenación de burbuja*: se efectúan tantos

pasos a través del archivo como sean necesarios, intercambiando elementos adyacentes; cuando en algún paso no se necesiten intercambios, el archivo estará ordenado. A continuación se da una implementación de este método.

```
void burbuja(tipoElemento a[], int N)
{
    int i, j;
    for (i = N; i >= 1; i--)
        for (j = 2; j >= i; j++)
            if (a[j-1] > a[j]) intercambio(a, j-1, j);
}
```

Es preciso un momento de reflexión antes de convencerse de que el programa hace lo que debe hacer: siempre que se encuentra el elemento de mayor valor durante el primer paso, se intercambia con cada elemento que queda a su derecha hasta que obtiene su posición en el extremo derecho del array. Después, en el segundo paso, se colocará en su posición definitiva el elemento con el segundo mayor valor, etc. Así, la ordenación de burbuja funciona como un tipo de ordenación por selección pero se debe trabajar mucho más para colocar a cada elemento en su posición definitiva.

Características del rendimiento de las ordenaciones elementales

Las Figuras 8.3, 8.4 y 8.5 proporcionan ilustraciones directas de las características operativas de las ordenaciones por selección, inserción y de burbuja. Estos diagramas muestran el contenido del array a para cada uno de los algoritmos después de que el bucle exterior se ha repetido $N/4$, $N/2$ y $3N/4$ veces (comenzando con una entrada formada por una permutación aleatoria de los números enteros del 1 al N). En los diagramas se coloca un cuadrado en la posición (i, j) cuando $a[i]=j$. Así, un array desordenado está representado por conjunto de cuadrados colocados aleatoriamente, mientras que en un array ordenado cada cuadrado aparecerá encima de aquel que está a su izquierda. Para mayor claridad en los diagramas, se representan las *permutaciones* (reordenaciones de los enteros del 1 al N), las que, al ordenarse, tienen todos los cuadrados alineados a lo largo de la diagonal principal. Los diagramas muestran cómo los diferentes métodos van avanzando hacia este objetivo.

La Figura 8.3 muestra cómo la ordenación por selección se mueve de izquierda a derecha, colocando los elementos en su posición definitiva sin tener que volver atrás. Lo que no es evidente a partir de este diagrama es el hecho de que la ordenación por selección emplea la mayoría de su tiempo en intentar encontrar el elemento mínimo en la parte «desordenada» del array.

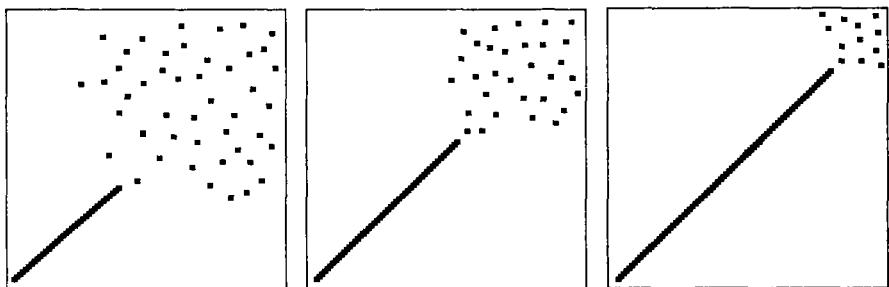


Figura 8.3 Ordenación por selección de una permutación aleatoria.

La Figura 8.4 muestra cómo la ordenación por inserción también se mueve de izquierda a derecha, insertando en su posición relativa los elementos que va encontrando, sin buscar más lejos. La parte izquierda del array está cambiando continuamente.

La Figura 8.5 muestra la similitud entre las ordenaciones por selección y de burbuja. Esta última «selecciona» el elemento máximo que queda en cada etapa, pero pierde algún tiempo poniendo orden en la parte «desordenada» del array.

Todos los métodos son cuadráticos, tanto en el peor caso como en el caso medio, y no necesitan memoria extra. Así, las comparaciones entre ellos dependen de la longitud de los bucles internos o de las características especiales de los datos de entrada.

Propiedad 8.1 *La ordenación por selección utiliza aproximadamente $N^2/2$ comparaciones y N intercambios.*

Esta propiedad es fácil de ver examinando la Figura 8.1, que es una tabla de dimensión $N \times N$ en la que a cada comparación le corresponde una letra. Pero esto representa aproximadamente la mitad de los elementos, precisamente los que están por encima de la diagonal. Cada uno de los $N - 1$ elementos que es-

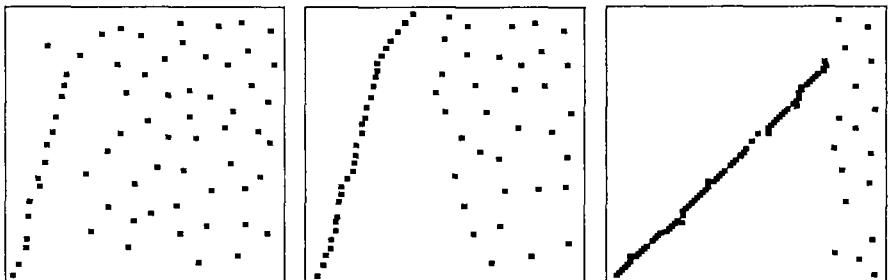


Figura 8.4 Ordenación por inserción de una permutación aleatoria.



Figura 8.5 Ordenación de burbuja de una permutación aleatoria.

tán en la diagonal (sin contar el último) corresponde a un intercambio. Con más precisión: para cada i desde 1 hasta $N - 1$, hay un intercambio y $N - i$ comparaciones, de forma que en total hay $N - 1$ intercambios y $(N - 1) + (N - 2) + \dots + 2 + 1 = N(N - 1)/2$ comparaciones. Estas observaciones son verdaderas para cualquier conjunto de datos de entrada: la única parte de la ordenación por selección que depende de dichos datos es el número de veces que se actualiza `min`. En el peor caso, esta cantidad podría ser también cuadrática, pero en el caso medio solamente pertenece a $O(N\log N)$, por lo que se puede afirmar que el tiempo de ejecución de una ordenación por selección es bastante insensible a los datos de entrada.■

Propiedad 8.2 *La ordenación por inserción utiliza aproximadamente $N^2/4$ comparaciones y $N^2/8$ intercambios en el caso medio y dos veces más en el peor caso.*

En la implementación anterior, el número de comparaciones y de «medios-intercambios» (desplazamientos) es el mismo. Como se acaba de exponer, esto se puede ver fácilmente en la Figura 8.2, que representa el diagrama de $N \times N$ con los detalles de las operaciones del algoritmo. Aquí se cuentan los elementos que quedan por debajo de la diagonal, todos ellos en el peor caso. Para una entrada aleatoria, es de esperar que cada elemento tenga que recorrer hacia atrás aproximadamente la mitad de las posiciones, por término medio, por lo que deberían contarse la mitad de los elementos que están por debajo de la diagonal. (No es difícil hacer que estos argumentos sean algo más rigurosos.)■

Propiedad 8.3 *Tanto en el caso medio como en el peor caso, la ordenación de burbuja utiliza aproximadamente $N^2/2$ comparaciones y $N^2/2$ intercambios.*

En el peor caso (archivo en orden inverso), está claro que en el i -ésimo paso de la ordenación de burbuja se necesitan $N - i$ comparaciones e intercambios, de forma que la demostración es como la de la ordenación por selección. Pero el tiempo de ejecución de la ordenación de burbuja depende de cómo estén orde-

nados los datos de entrada. Por ejemplo, se observa que si el archivo ya está ordenado sólo se necesita un paso (la ordenación por inserción también es rápida en este caso). En cambio, el rendimiento en el caso medio no es significativamente mejor que en el peor caso, pero en estas condiciones este análisis es bastante más difícil.■

Propiedad 8.4 *La ordenación por inserción es lineal para los archivos «casi ordenados».*

Aunque el concepto de archivo «casi ordenado» es necesariamente bastante impreciso, la ordenación por inserción funciona bien con algunos tipos de archivos no aleatorios que aparecen con frecuencia en la práctica. Normalmente se abusa de las ordenaciones de aplicación general al utilizarlas en estas situaciones; en realidad, la ordenación por inserción puede aprovechar el orden que presente el archivo.

Por ejemplo, considérese la operación de ordenar por inserción un archivo que ya está ordenado. De forma inmediata se determina que cada elemento está en el lugar que le corresponde en el archivo y el tiempo de ejecución total es lineal. Lo mismo ocurre con la ordenación de burbuja, pero la ordenación por selección todavía es cuadrática. Incluso si el archivo no está completamente ordenado, la ordenación por inserción puede resultar bastante útil porque el tiempo de ejecución tiene una dependencia bastante fuerte del orden que tenga ya el archivo. El tiempo de ejecución depende del número de *inversiones*: para cada elemento se cuenta el número de elementos superiores a él, de los que quedan a su izquierda. Ésta es precisamente la distancia que tienen que recorrer los elementos cuando se insertan en el archivo durante la ordenación por inserción. Un archivo que esté algo ordenado tendrá menos inversiones que otro que esté arbitrariamente desordenado.

Si se desea añadir algunos elementos a un archivo ordenado para obtener un archivo ordenado más grande, una forma de hacerlo consiste en añadir los nuevos elementos al final del archivo y a continuación llamar a un algoritmo de ordenación. Claro está, el número de inversiones será bajo: un archivo que tiene únicamente un número constante de elementos sin ordenar tendrá un número lineal de inversiones. Otro ejemplo es el de un archivo en el que cada elemento está solamente a una cierta distancia constante de su posición definitiva. Tales archivos aparecen a veces en las etapas iniciales de algunos métodos de ordenación avanzados: en un determinado momento merece la pena cambiar a la ordenación por inserción.

Para estos archivos, la ordenación por inserción superará incluso a los métodos sofisticados que se verán en los siguientes capítulos.■

Para comparar los métodos con más profundidad, se necesita analizar el coste de las comparaciones y de los intercambios, factores que dependen tanto del tamaño de los registros como de las claves. Por ejemplo, si los registros tienen claves de una palabra, como en las implementaciones anteriores, entonces un intercambio (dos accesos al array) cuesta aproximadamente el doble que una

comparación. En estas circunstancias, el tiempo de ejecución de una ordenación por selección y el de una por inserción son prácticamente iguales, pero la ordenación de burbuja es dos veces más lenta. (De hecho, ¡la ordenación de burbuja es dos veces más lenta que la de inserción, casi bajo cualquier circunstancia!) Pero si los registros son grandes en comparación con las claves, entonces será mejor la ordenación por selección.

Propiedad 8.5 *La ordenación por selección es lineal para archivos con registros grandes y claves pequeñas.*

Supóngase que los costes de una comparación y de un intercambio son de 1 y de M unidades de tiempo, respectivamente (éste puede ser el caso, por ejemplo, de registros de M palabras y claves de una), entonces, la ordenación por selección de un archivo de tamaño NM lleva aproximadamente N^2 unidades de tiempo para las comparaciones y alrededor de NM para los intercambios. Si $N = O(M)$, esto es un tiempo lineal en el tamaño de los datos. ■

Ordenación de archivos con grandes registros

En realidad, es posible (y deseable) arreglar las cosas para que *cualquier* método de ordenación utilice sólo N «intercambios» de registros completos, haciendo que el algoritmo opere indirectamente sobre el archivo (utilizando un array de índices) y reordenándolo después.

Específicamente, si el array $a[1], \dots, a[N]$ contiene registros grandes, es preferible manipular un «array de índices» $p[1], \dots, p[N]$ que accede al array original sólo para las comparaciones. Si inicialmente se define $p[i] = i$, sólo se necesitará modificar los algoritmos anteriores (y todos los de los capítulos siguientes) para que hagan referencia a $a[p[i]]$ en lugar de a $a[i]$ al utilizar $a[i]$ en una comparación, y para hacer referencia a p en lugar de a a cuando se hagan desplazamientos de datos. Esto produce un algoritmo que «ordenará» el array de índices de manera que $p[1]$ es el índice del elemento más pequeño de a , $p[2]$ es el índice del segundo elemento más pequeño de a , etc., y así se evitará el coste de mover excesivamente grandes registros. El programa siguiente muestra cómo puede modificarse la ordenación por inserción para que funcione de esta manera.

```
void insercion(tipoElemento a[], int p[], int N)
{
    int i, j; tipoElemento v;
    for (i = 0; i <= N; i++) p[i] = i ;
    for (i = 2; i <= N; i++)
    {
        j = i - 1;
        v = a[p[j]];
        while (j > 0 && a[p[j-1]] > v)
        {
            a[p[j]] = a[p[j-1]];
            p[p[j]] = j;
            j--;
        }
        a[p[j]] = v;
        p[p[j]] = i;
    }
}
```

```

v = p[i]; j = i;
while (a[p[j-1]] > a[v] )
    { p[j] = p [j-1]; j--; }
p[j] = v;
}
}

```

En este programa se accede al array a solamente para comparar las claves de dos registros. Así, podría modificarse fácilmente para tratar archivos con registros muy grandes cambiando la comparación para que acceda sólo a un campo pequeño de un gran registro, o haciendo el proceso de comparación algo más complicado. La Figura 8.6 muestra cómo este procedimiento produce una permutación que especifica el orden en que podría accederse a los elementos del array para definir una lista ordenada. Para muchas aplicaciones, esto será suficiente (no se necesitará desplazar totalmente los datos). Por ejemplo, se podrían imprimir los datos ordenados haciendo referencia a cada uno por medio del array de índices, como en la propia ordenación.

**Antes de la
ordenación**

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a [k]	E	J	E	M	P	L	O	A	O	R	D	E	N	A	R
p [k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

**Después de la
ordenación**

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a [k]	E	J	E	M	P	L	O	A	O	R	D	E	N	A	R
p [k]	8	14	11	1	3	12	2	6	4	13	7	9	5	10	15

**Después de la
permutación**

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a [k]	A	A	D	E	E	E	J	L	M	N	O	O	P	R	R
p [k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figura 8.6 Reorganización de un array «ordenado».

Pero ¿qué ocurre si realmente se deben reorganizar los datos, como es el caso de la parte de abajo de la Figura 8.6? Si se dispone de suficiente memoria extra para hacer otra copia del array, esto es trivial, pero ¿qué hacer en la situación más normal, cuando no se dispone de espacio suficiente para hacer otra copia del archivo?

En el ejemplo, lo primero que se debería hacer es poner el registro que tenga la clave más pequeña (el que corresponde al índice $p[1]$) en la primera posición en el archivo. Pero, antes de hacerlo, se necesita guardar el registro que está en esa posición, por ejemplo en t . Ahora, después del movimiento, se puede considerar que hay un «hueco» en el archivo en la posición $p[1]$, pero se sabe que el registro de la posición $p[p[1]]$ llenará finalmente este hueco. Continuando

de esta manera, llegará el momento en que llene el hueco el elemento original de la primera posición, que se había almacenado en t . En el caso del ejemplo este proceso conduce a la serie de asignaciones $t=a[1]$; $a[1]=a[8]$; $a[8]=a[6]$; $a[6]=a[12]$; $a[12]=a[9]$; $a[9]=a[4]$; $a[4]=a[1]$; $a[1]=t$. Estas asignaciones colocan a los registros con claves A, E, E, L, M y O en su lugar adecuado dentro del archivo, lo que puede indicarse poniendo $p[1]=1$, $p[8]=8$, $p[6]=6$, $p[12]=12$, $p[9]=9$ y $p[4]=4$. (Cualquier elemento con $p[i]=i$ está en su lugar definitivo y no es necesario volver a tocarlo.) Ahora, se puede reanudar el proceso para el siguiente elemento que no esté en su lugar, y así sucesivamente, hasta que por último se reorganice todo el archivo, moviendo cada registro sólo una vez, como se muestra en el siguiente código:

```
void insitu(tipoElemento a[], int p[], int N)
{
    int i, j, k; tipoElemento t;
    for (i = 1; i <= N; i++)
        if (p[i] != i)
    {
        t = a[i]; k = i;
        do
        {
            j = k; a[j] = a[p[j]];
            k = p[j]; p[j] = j;
        }
        while (k != i);
        a[j] = t;
    }
}
```

Por supuesto, la viabilidad de esta técnica para aplicaciones particulares depende del tamaño relativo de los registros y de las claves del archivo a ordenar. Por cierto, no se debería utilizar con un archivo de registros pequeños, porque se necesita mucho espacio extra para el array de índices y mucho tiempo extra para las comparaciones indirectas. Pero para archivos que contienen registros grandes, casi siempre es preferible utilizar una ordenación indirecta y, en muchas aplicaciones, no es necesario desplazar todos los datos. Por supuesto, como se vio anteriormente, para archivos que tienen registros muy grandes el método a utilizar es la ordenación por selección.

La técnica anterior de «array de índices» funcionará en cualquier lenguaje de programación que cuente con arrays. En C++ suele ser conveniente desarrollar una implementación basada en el mismo principio, utilizando las direcciones de máquina de los elementos del array (los «auténticos punteros» que se presentaron brevemente en el Capítulo 3). Por ejemplo, el código siguiente implementa una ordenación por inserción utilizando un array p de punteros:

```

void insercion(tipoElemento a[], tipoElemento *p[], int N)
{
    int i, j; tipoElemento *v;
    for (i = 0; i <= N; i++) p[i] = &a[i];
    for (i = 2; i <= N; i++)
    {
        v = p[i]; j = i;
        while (*p[j-1] > *v)
            { p[j] = p[j-1]; j--; }
        p[j] = v;
    }
}

```

Una de las características fundamentales que C++ ha heredado de C es la fuerte relación existente entre punteros y arrays. En general, los programas implementados con punteros son más eficaces, pero más difíciles de entender (aunque para alguna aplicación concreta no hay mucha diferencia). El lector que esté interesado puede implementar el programa *insitu* necesario para la ordenación por punteros anterior.

En las implementaciones de este libro normalmente se accederá de manera directa a los datos, aun sabiendo que se podrían utilizar punteros o arrays de índices para evitar realizar un número excesivo de desplazamientos de datos al hacer las ordenaciones. Debido a la existencia de esta ordenación indirecta, las conclusiones que se muestran en este capítulo y en los que siguen, cuando se comparan los métodos para ordenar archivos de enteros, se pueden aplicar a situaciones más generales.

Ordenación de Shell

La ordenación por inserción es lenta porque únicamente se realizan intercambios entre elementos adyacentes. Por ejemplo, si el elemento más pequeño está al final del array, se necesitan N pasos para situarlo en su lugar correspondiente. *La ordenación de Shell (Shellsort)* es una simple generalización de la ordenación por inserción en la que se gana rapidez al permitir el intercambio entre elementos que están muy alejados.

La idea es reorganizar el archivo para que tenga la propiedad de que, tomando todos los elementos h -ésimos (comenzando por cualquier sitio), se obtenga un archivo ordenado. Tal archivo se dice que está *h-ordenado*. Diciéndolo de otra forma, un archivo *h*-ordenado está constituido por h archivos ordenados independientes, entrelazados entre sí. *H*-ordenando el archivo para algunos valores grandes de h se pueden intercambiar elementos muy distantes en el array,

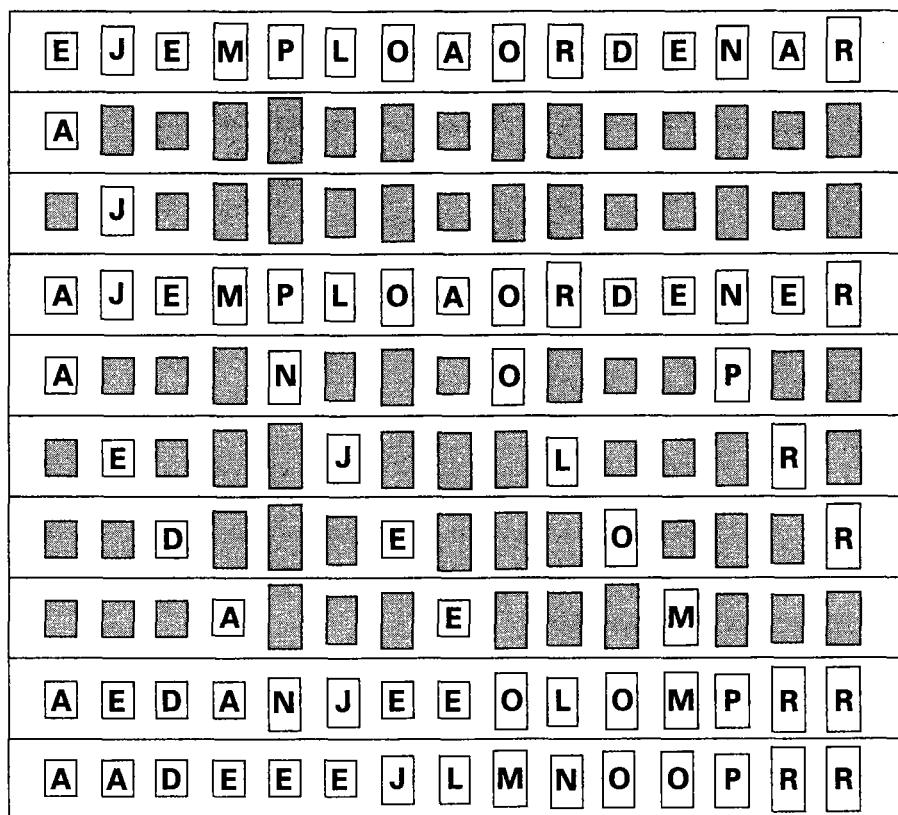


Figura 8.7 Ordenación de Shell.

y así se facilita la h -ordenación para pequeños valores de h . Utilizando este procedimiento para cualquier serie decreciente de valores de h que termine en 1 se genera un archivo ordenado: éste es el principio de la ordenación de Shell.

La Figura 8.7 muestra la operación de la ordenación de Shell para el archivo ejemplo con los incrementos decrecientes ..., 13, 4, 1. En el primer paso, la E está en la posición 1, y se compara (y se intercambia) con la A de la posición 14, y después la J de la posición 2 se compara con la R de la posición 15. En el segundo paso se reordenan las letras A P O N de las posiciones 1, 5, 9 y 13 poniendo en esas posiciones A N O P, después las letras J L R E de las posiciones 2, 6, 10 y 14, poniendo en ellas E J L R, y así sucesivamente. El último paso es precisamente la ordenación por inserción; pero ningún elemento tiene que desplazarse muy lejos.

Una forma de implementar la ordenación de Shell podría ser utilizar, para cada h , la ordenación por inserción de forma independiente en cada uno de los h subarchivos. (No deberían utilizarse centinelas porque se necesitaría un gran número de ellos para los mayores valores de h .) Pero, en cambio, se puede hacer

más fácil todavía: Si se reemplaza cada aparición de «1» por « h » (y de «2» por « $h+1$ ») en la ordenación por inserción, el programa resultante h -ordena el archivo y se obtiene una implementación más compacta de la ordenación de Shell, como se indica a continuación:

```
void ordenshell(tipoElemento a[],int N)
{
    int i, j, h; tipoElemento v;
    for (h = 1; h <= N/9; h = 3*h+1) ;
    for ( ; h > 0; h /= 3)
        for (i = h+1; i <= N; i += 1)
        {
            v = a[i]; j = i;
            while (j > h && a[j-h]>v)
                { a[j] = a[j-h]; j -= h; }
            a[j] = v;
        }
}
```

Este programa utiliza la serie de incrementos decrecientes ..., 1093, 364, 121, 40, 13, 4, 1. En la práctica, otra serie podría resultar tan buena como ésta pero, como se indica a continuación, hay que tener un poco de cuidado al elegirla. La Figura 8.8 muestra cómo actúa este programa ante una permutación aleatoria, mostrando el contenido del array a después de cada h -ordenación.

La sucesión de incrementos decrecientes de este programa es fácil de utilizar y conduce a una ordenación eficaz. Hay otras muchas sucesiones que conducen a ordenaciones mejores (el lector puede entretenerte intentando descubrir una), pero es difícil mejorar el programa anterior en más de un 20%, incluso para N relativamente grande. (Sin embargo, la posibilidad de que existan sucesiones mucho mejores sigue siendo bastante real.) Por el contrario, existen algunas sucesiones más desfavorables: por ejemplo, ..., 64, 32, 16, 8, 4, 2, 1 conduce a un mal rendimiento porque los elementos que están en las posiciones impares no se comparan con los elementos que están en las posiciones pares hasta el final. De forma similar, algunas veces la ordenación de Shell se implementa comenzando por $h=N$ (en lugar de inicializarse de manera que siempre se utilice la misma sucesión de antes). Virtualmente, esto asegura que surgirá una sucesión desfavorable para algún N .

La anterior descripción de la eficacia de la ordenación de Shell es imprecisa por necesidad, porque nadie ha sido capaz de analizar el algoritmo. Esto hace que no sólo sea difícil evaluar las diferentes series de incrementos, sino también comparar analíticamente la ordenación de Shell con otros métodos. Ni siquiera se conoce la forma funcional del tiempo de ejecución para esta ordenación (como mucho se sabe que depende de la sucesión de incrementos). Para el programa anterior podrían darse dos conjeturas: $N(\log N)^2$ y $N^{1.25}$. El tiempo de ejecución

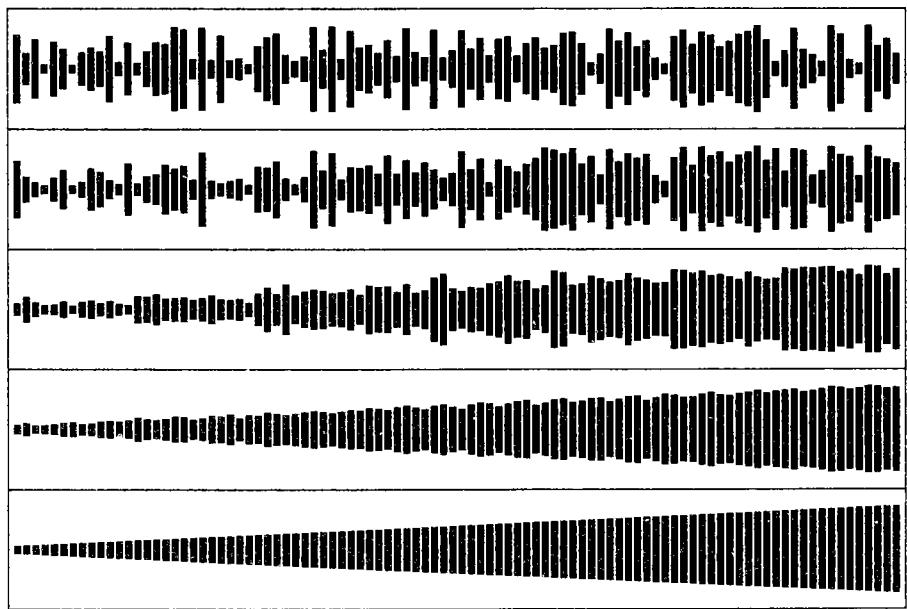


Figura 8.8. Ordenación de Shell para una permutación aleatoria.

no es particularmente sensible al orden inicial del archivo, en especial en contraste con, por ejemplo, la ordenación por inserción, en la que el tiempo de ejecución es lineal para un archivo ya ordenado y cuadrático para un archivo en orden inverso. La Figura 8.9 muestra las operaciones de la ordenación de Shell en un archivo de este tipo.

Propiedad 8.6 *La ordenación de Shell nunca hace más de $N^{3/2}$ comparaciones (para los incrementos 1, 4, 13, 40, 121,...).*

La demostración de esta propiedad está más allá del alcance de este libro; pero, además de apreciar su dificultad, el lector también puede convencerse de que la ordenación de Shell se comporta bien en la práctica intentando construir un archivo en el que la ordenación de Shell se ejecute lentamente. Como se mencionó antes, existen algunas sucesiones de incrementos desfavorables para las que la ordenación de Shell puede necesitar un número cuadrático de comparaciones, pero se ha demostrado que la cota $N^{3/2}$ es válida para una amplia variedad de sucesiones, como la que se ha utilizado con anterioridad. Incluso se conocen mejores cotas para el peor caso de algunas sucesiones especiales.■

La Figura 8.10 muestra una visión diferente de las operaciones que realiza la ordenación de Shell, comparable a la de las Figuras 8.3, 8.4 y 8.5. Esta figura presenta el contenido del array después de cada h -ordenación (excepto la última, que es la que completa la ordenación). En estos diagramas podría imagi-

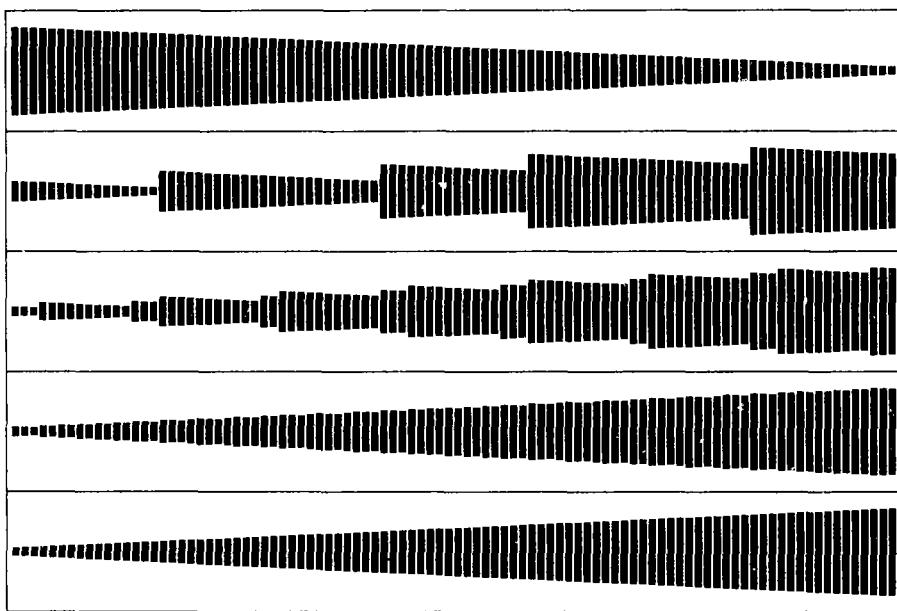


Figura 8.9. Ordenación de Shell para una permutación en orden inverso.

narse una goma elástica fija en las esquinas inferior izquierda y superior derecha, que se estira y ajusta para llevar todos los puntos hacia la diagonal. Cada uno de los tres diagramas de las Figuras 8.3, 8.4 y 8.5 representa el hecho de que cada algoritmo que se muestra debe realizar una cantidad de trabajo significativa; por el contrario, cada uno de los diagramas de la Figura 8.10 representa sólo un paso de h -ordenación.

La ordenación de Shell es el método elegido en muchas aplicaciones, ya que su tiempo de ejecución es aceptable, incluso para archivos moderadamente grandes (por ejemplo, con menos de 5.000 elementos) y únicamente se necesita

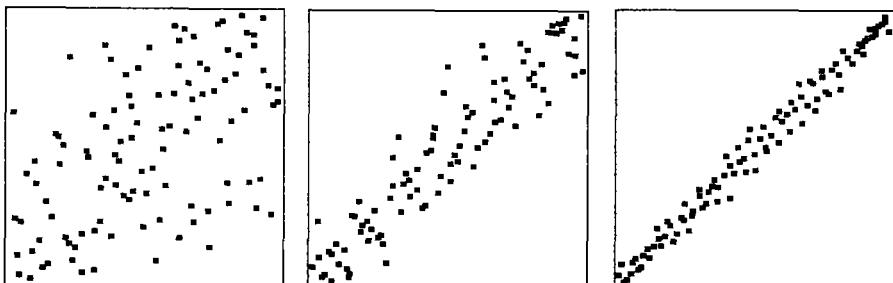


Figura 8.10. Ordenación de Shell de una permutación aleatoria.

un pequeño código, fácil de ejecutar. En los siguientes capítulos se verán métodos que son más eficaces, pero que sólo son el doble de rápidos (cuando mucho), excepto para grandes valores de N , y son bastante más complicados. En resumen, si se tiene un problema de ordenación, lo mejor es *utilizar el programa anterior*, y determinar después si vale la pena el esfuerzo extra que se necesita para cambiarlo por un método más sofisticado.

Cuenta de distribuciones

Hay una situación muy especial para la que existe un sencillo algoritmo de ordenación: «ordenar un archivo de N registros cuyas claves son distintos números enteros entre 1 y N .» Este problema puede resolverse utilizando un array temporal b con la sentencia `for (i = 1; i <= N; i++) b[a[i]] = a[i]`. (O, como se vio anteriormente, es posible, aunque bastante difícil, resolver este problema sin un array auxiliar.)

Un problema más realista, pero con el mismo espíritu, consiste en «ordenar un archivo de N registros cuyas claves son números enteros entre 0 y $M-1$.» Si M no es demasiado grande, se puede utilizar para resolver este problema un algoritmo denominado *cuenta de distribuciones*. La idea es contar el número de claves de cada valor y después utilizar los números que se han contado para desplazar los registros hacia su posición durante un segundo recorrido a través del archivo, como se indica en el código siguiente:

```
for (j = 0; j < M; j++) contador[j] = 0;
for (i = 1; i <= N; i++) contador[a[i]]++;
for (j = 1; j < M; j++) contador[j] += contador[j-1];
for (i = N; i >= 1; i--) b[contador[a[i]]--] = a[i];
for (i = 1; i <= N; i++) a[i] = b[i];
```

Para ver cómo funciona este código, se considera el archivo de ejemplo formado por los números enteros de la fila superior de la Figura 8.11. El primer bucle `for` inicializa el contador a 0; el segundo pone $\text{contador}[1]=6$, $\text{contador}[2]=4$, $\text{contador}[3]=1$, y $\text{contador}[4]=3$ ya que hay seis letras A, cuatro B, etc. A continuación el tercer bucle `for` acumula estos números y se obtiene $\text{contador}[1]=6$, $\text{contador}[2]=10$, $\text{contador}[3]=11$, y $\text{contador}[4]=15$. Esto es, hay seis claves inferiores o iguales que A, diez claves menores o iguales que B, etcétera.

Ahora, los contadores pueden servir de índices para ordenar el array, como se muestra en la figura. El array original a se muestra en la línea superior; el resto de la figura muestra el array temporal que se está rellenando. Por ejemplo, cuando se encuentre la A al final del archivo, se colocará en la posición 6, ya que $\text{contador}[1]$ indica que hay seis claves menores o iguales que A. Después

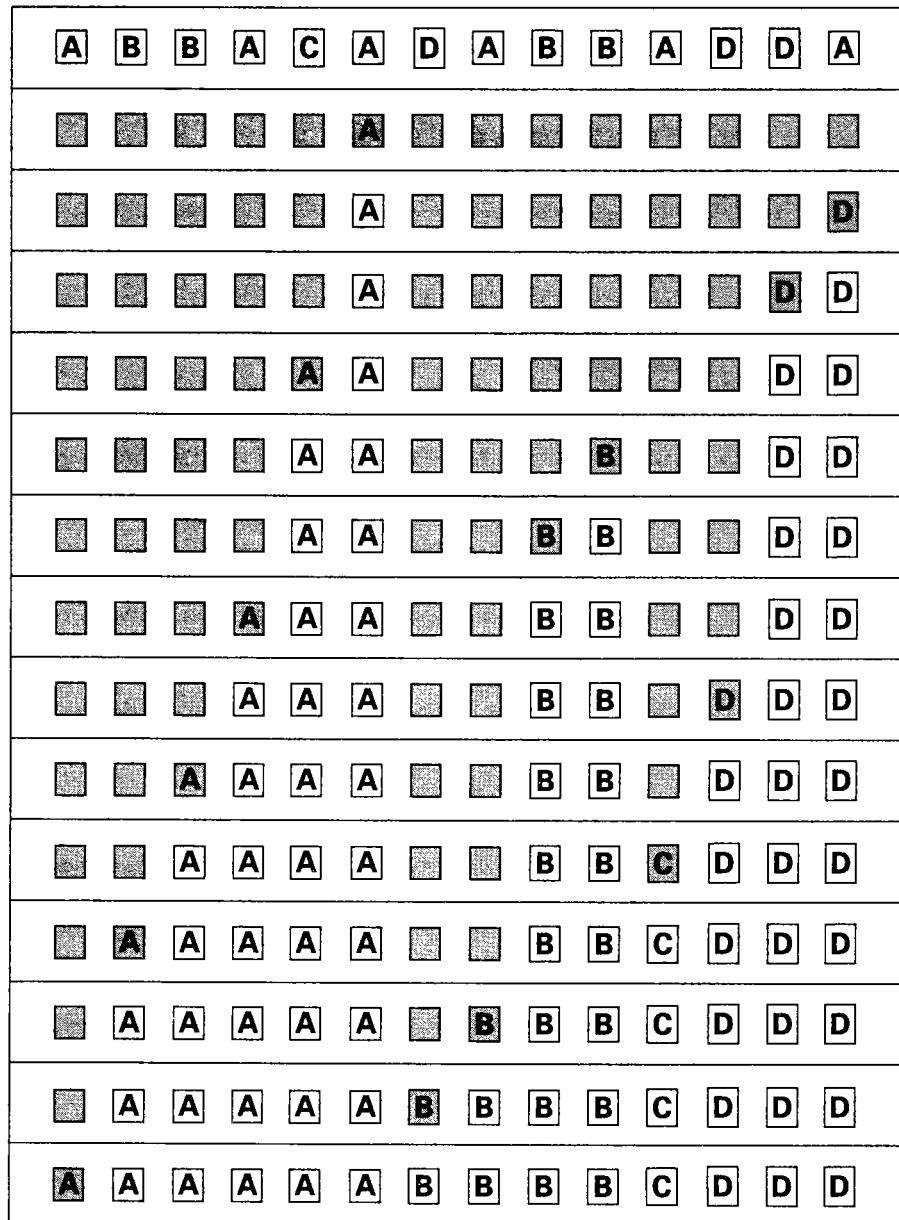


Figura 8.11. Cuenta de distribuciones.

contador[1] se reduce en una unidad, de manera que ahora hay una clave menos entre las menores o iguales que A. A continuación, la D de la penúltima posición del archivo se coloca en la posición 14 y contador[4] se reduce en una unidad, etc. El bucle interno se realiza desde N hasta 1, de modo que la ordenación será estable. (El lector puede intentar comprobar esta afirmación.)

Este método funcionará muy bien para los tipos de archivos descritos anteriormente. Además, se puede utilizar para obtener métodos mucho más potentes que se examinarán en el Capítulo 10.

Ejercicios

1. Dar una sucesión de operaciones «comparar-intercambiar» para ordenar cuatro registros.
2. ¿Cuál de los tres métodos elementales (ordenación por selección, por inserción, de burbuja) es más rápido en un archivo que ya está ordenado?
3. ¿Cuál de los tres métodos elementales es más rápido para un archivo que está en orden inverso?
4. Comprobar la hipótesis de que la ordenación por selección es el más rápido de los métodos elementales (para ordenar números enteros), seguida por la inserción y después por la ordenación de burbuja.
5. Dar una buena razón de por qué puede no ser conveniente utilizar una clave centinela en la ordenación por inserción (aparte de la que se dio en la implementación de la ordenación de Shell).
6. ¿Cuántas comparaciones utilizará la ordenación de Shell para hacer una 7-ordenación, y después una 3-ordenación de las claves C U E S T I O N F A C I L?
7. Dar un ejemplo para mostrar por qué 8, 4, 2, 1 no sería una buena forma de finalizar una sucesión de incrementos para hacer una ordenación de Shell.
8. ¿Es estable la ordenación por selección? ¿Y la ordenación por inserción? ¿Y la ordenación de burbuja?
9. Dar una versión especializada de la cuenta de distribuciones para ordenar archivos donde los elementos sólo pueden tomar dos valores (o bien x o bien y).
10. Experimentar con diferentes sucesiones de incrementos para la ordenación de Shell: encontrar una que sea más rápida que la que se dio para un archivo aleatorio de 1.000 elementos.

Quicksort

En este capítulo se estudiará el algoritmo de ordenación que es probablemente el más utilizado de todos: la ordenación rápida (*Quicksort*). El algoritmo básico fue inventado en 1960 por C.A.R. Hoare, y desde entonces ha sido objeto de numerosos estudios. El Quicksort es popular porque no es difícil de implementar, proporciona unos buenos resultados generales (funciona bien en una amplia diversidad de situaciones) y en muchos casos consume menos recursos que cualquier otro método de ordenación.

Entre las ventajas del algoritmo de ordenación rápida destacan: trabaja *in situ* (utiliza sólo una pequeña pila auxiliar), necesita solamente del orden de $N \log N$ operaciones en promedio para ordenar N elementos y tiene un bucle interno extremadamente corto. Los inconvenientes son que es recursivo (si no se puede utilizar la recursión la implementación es complicada), que en el peor caso necesita aproximadamente N^2 operaciones y que es frágil: si durante la implementación pasa inadvertido un simple error, puede causar un mal comportamiento en ciertos archivos.

El rendimiento del Quicksort se entiende muy bien. Ha sido objeto de minuciosos análisis matemáticos y se puede describir con precisión. El análisis ha sido comprobado por una extensa experiencia empírica y el algoritmo se ha refinado hasta el punto de convertirse en el método elegido en una gran variedad de aplicaciones prácticas de ordenación. Esto hace que merezca la pena estudiarlo con más cuidado que otros algoritmos, con el fin de implementar de manera eficaz el Quicksort. Existen técnicas de implementación similares que son apropiadas para otros algoritmos y que pueden utilizarse con la ordenación rápida porque así se comprende mejor su rendimiento.

Es muy tentador tratar de desarrollar formas de mejorar el Quicksort: encontrar un algoritmo de ordenación más rápido es una de las utopías de la informática. Casi desde el momento en que Hoare hizo público el algoritmo han ido apareciendo en los libros versiones «mejoradas» del mismo. Se han intentado y analizado muchas ideas, pero es fácil decepcionarse porque este algoritmo está tan bien equilibrado que los efectos de las mejoras en una parte del

programa pueden estar más que compensados por las consecuencias de un mal rendimiento en otra. En este capítulo se examinarán con algún detalle tres modificaciones que mejoran sustancialmente el Quicksort.

Una versión afinada con cuidado del Quicksort es probable que se ejecute más rápidamente en la mayoría de las computadoras que cualquier otro método de ordenación. De cualquier forma, debe tenerse en cuenta que la optimización de cualquier algoritmo puede hacerlo más frágil, conduciendo a efectos indeseables e inesperados para ciertos datos de entrada. Una vez que se ha desarrollado una versión que parezca estar libre de tales efectos, es probablemente ésta la que se debería tener como una de las utilidades de ordenación de una biblioteca o en una aplicación de ordenación seria. Pero si no se está dispuesto a realizar un esfuerzo adicional para poner a punto una implementación del Quicksort que resulte correcta, la ordenación de Shell podría resultar una elección segura que funcionará bastante bien con un menor esfuerzo de implementación.

El algoritmo básico

El Quicksort es un método de ordenación de «divide y vencerás». Funciona *dividiendo* un archivo en dos partes, y ordenando independientemente cada una de ellas. Como se verá, el punto exacto de la partición depende del archivo, y así el algoritmo presenta la siguiente estructura recursiva:

```
void ordenrapido(tipoElemento a[], int izq, int der)
{
    int i;
    if (r > izq)
    {
        i = particion (a, izq, der);
        ordenrapido(a, izq, i-1);
        ordenrapido(a, i+1, der);
    }
}
```

Los parámetros *izq* y *der* delimitan el subarchivo del archivo original que se va a ordenar; la llamada a *ordenrapido(a, 1, N)* ordena el archivo en su totalidad.

Lo esencial del método es el procedimiento *particion*, que debe reordenar el array para que se verifiquen las siguientes condiciones:

- (i) el elemento *a[i]* está en su lugar definitivo en el array para algún *i*,
- (ii) ninguno de los elementos de *a[izq], ..., a[i-1]*, son mayores que *a[i]*,

(iii) ninguno de los elementos de $a[i+1], \dots, a[der]$ son menores que $a[i]$.

Esto se puede implementar de una manera muy fácil y sencilla mediante la siguiente estrategia general. En primer lugar, elegir $a[der]$ de manera arbitraria como el elemento que irá en su posición definitiva. Después, explorar el array de izquierda a derecha hasta encontrar un elemento mayor que $a[der]$, y volverlo a explorar de derecha a izquierda hasta encontrar un elemento menor que $a[der]$. Los dos elementos en los que se detiene el proceso están obviamente mal situados en el array dividido resultante y por tanto se intercambian. (En realidad, por razones que se darán posteriormente, es mejor detener también las exploraciones en los elementos iguales a $a[der]$, aunque parezca que al hacerlo así se van a realizar algunos intercambios innecesarios). Continuando de esta forma se tiene la seguridad de que todos los elementos del array situados a la izquierda del puntero izquierdo son menores que $a[der]$ y de que todos los situados a la derecha del puntero derecho son mayores que $a[der]$. Cuando los punteros de exploración se cruzan, el proceso de partición está casi acabado: todo lo que queda por hacer es intercambiar $a[der]$ con el elemento que está más a la izquierda del subarchivo derecho (el elemento apuntado por el puntero izquierdo).

La Figura 9.1 muestra cómo se divide con este método el archivo ejemplo de claves. Se elige como elemento de partición al elemento que está más a la derecha, R. Al principio la exploración desde la izquierda se para en la R y a continuación la exploración desde la derecha se para en la A (como se muestra en la segunda línea de la tabla) y entonces se intercambian estas dos letras. A continuación, como los punteros se cruzan, la exploración desde la izquierda se para en la R, mientras que desde la derecha se detiene en la N. El movimiento apropiado en este caso consiste en intercambiar la R de la derecha con la otra R, dejando el archivo dividido tal y como se muestra en la última línea de la Figura 9.1.

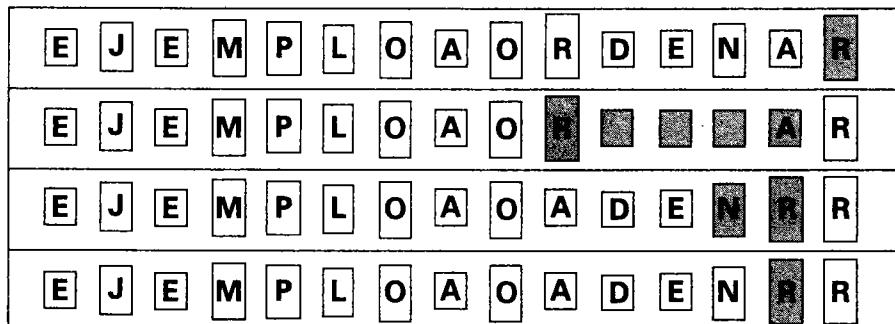


Figura 9.1 Operación de partición.

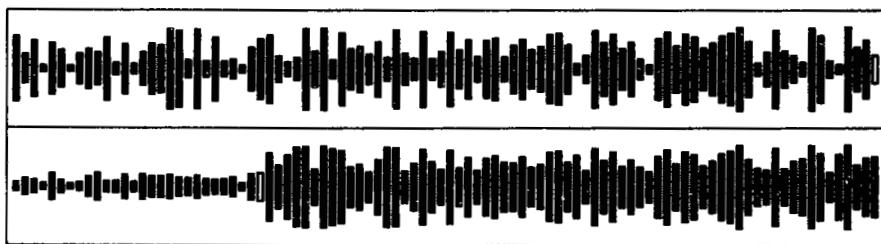


Figura 9.2 Partición de un archivo mayor.

El proceso de partición no es estable puesto que durante cualquier intercambio toda clave podría desplazarse detrás de un gran número de claves iguales a ella (que aún no se han examinado).

La Figura 9.2 muestra el resultado de dividir un archivo más grande: con los elementos pequeños a la izquierda y los grandes a la derecha, el archivo dividido presenta considerablemente más «orden» que el archivo aleatorio. La ordenación se termina ordenando los dos subarchivos que quedan a cada lado del elemento de partición (recursivamente). El siguiente programa proporciona una implementación completa del método.

```
void ordenrapido(tipoElemento a[], int izq, int der)
{
    int i, j; tipoElemento v;
    if (der > izq)
    {
        v = a[der]; i = izq-1; j = der;
        for (;;)
        {
            while (a[++i] < v) ;
            while (a[--j] > v) ;
            if (i >= j) break;
            intercambio(a, i, j);
        }
        intercambio(a, i, der);
        ordenrapido(a, izq, i-1);
        ordenrapido(a, i+1, der);
    }
}
```

En esta implementación, la variable *v* contiene el valor actual del «elemento de partición» *a[der]*, con *i* y *j* como los punteros izquierdo y derecho respectivamente. El bucle de la partición se implementa como un bucle infinito, con

un break de salida cuando se cruzan los punteros. Este método es realmente la demostración típica de por qué se utiliza la capacidad break: el lector podría entretenerte considerando cómo se puede implementar el método de partición sin utilizar break.

Como en la ordenación por inserción, se necesita una «clave centinela» para detener la exploración cuando el elemento de partición sea el más pequeño del archivo. En esta implementación no se necesita ningún centinela para detener la exploración cuando el elemento de partición sea el más grande del archivo, porque es él mismo quien la detiene en el lado derecho del archivo. Pronto se verá una forma sencilla de eliminar ambas claves centinela.

El «bucle interno» de la ordenación rápida implica simplemente incrementar un puntero y comparar un elemento del array con un valor fijo. Esto es lo que realmente hace rápido a este método de ordenación: es difícil imaginar un bucle interno más sencillo. También aquí se encuentra una prueba del efecto beneficioso de las claves centinela, puesto que añadir una comprobación superflua al bucle interno tiene un gran efecto en el rendimiento.

La ordenación termina ordenando recursivamente los dos subarchivos. La Figura 9.3 muestra estas llamadas recursivas. Cada línea representa el resultado de dividir el subarchivo, así como el elemento de partición elegido (sombreado en el diagrama). Si la primera comprobación del programa fuera $\text{der} >= \text{izq}$ en lugar de $\text{der} > \text{izq}$, cada elemento acabaría por utilizarse como elemento de partición para poderse colocar en su posición final; en la implementación dada, los archivos de tamaño 1 no se dividen, como se puede ver en la Figura 9.3. Más adelante se estudiará una generalización de esta mejora.

La característica más negativa del programa anterior es que para archivos sencillos es muy ineficaz. Por ejemplo, si se le llama para un archivo que ya está ordenado, las particiones serían degeneradas y el programa se llamaría a sí mismo N veces, quitando sólo un elemento en cada llamada. Esto significa no sólo que el tiempo requerido será del orden de $N^2/2$, sino que además la memoria necesaria para solventar la recursión será del orden de N (como se verá más adelante), lo cual es inaceptable. Por fortuna hay formas relativamente sencillas de evitar que este caso tan negativo pueda ocurrir en las implementaciones reales del programa.

Cuando hay claves iguales en el archivo, hay dos detalles aparentemente poco importantes, pero que en realidad sí lo son. En primer lugar se plantea la pregunta de si ambos punteros se han de detener en las claves iguales al elemento de partición o si uno debe parar y el otro continuar la exploración o si ambos deben continuar. Esta cuestión se ha estudiado matemáticamente en detalle y los resultados muestran que es mejor que se detengan los dos punteros. Esto tiende a equilibrar las particiones cuando hay muchas claves iguales. Segundo, se plantea la cuestión de solucionar adecuadamente el cruce de punteros cuando hay claves iguales. De hecho, el programa anterior puede mejorarse ligeramente terminando la exploración cuando $j < i$ y utilizar ordenrapido (a, izq, j) para la primera llamada recursiva. Esto es una mejora porque cuando $j=i$ se pueden poner dos elementos en su posición dejando que el bucle se ejecute una

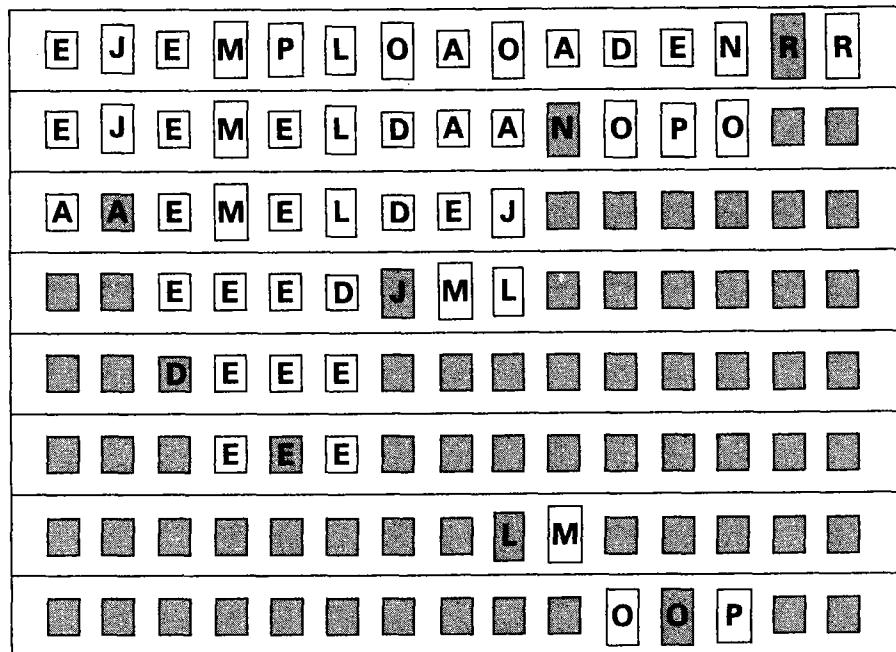


Figura 9.3 Subarchivos en el Quicksort.

vez más. (Este caso ocurriría, por ejemplo, si R fuera E en el ejemplo anterior.) Probablemente merezca la pena hacer este cambio porque el programa, tal y como se ha dado, deja un registro con una clave igual a la clave de partición en `a[der]`, y esto hace una primera partición degenerada en la llamada ordenamiento rápido (`a, i+1, der`) porque la clave situada más a la derecha es también la más pequeña. La implementación del método de partición dado anteriormente es algo más fácil de comprender, por lo que será la que se trate en la siguiente presentación. Sin embargo, hay que ser consciente de que esta modificación debería realizarse cuando haya un gran número de claves iguales.

Características de rendimiento del Quicksort

Lo mejor que podría ocurrir en la ordenación rápida sería que en cada etapa de partición se dividiera el archivo exactamente por la mitad. Esto haría que el número de comparaciones a utilizar por la ordenación rápida satisficiera la recurrencia del método divide y vencerás

$$C_N = 2C_{N/2} + N.$$

El término $2C_{N/2}$ cubre el coste de ordenación de los dos subarchivos; el término N es el coste de examinar cada elemento, utilizando un puntero de partición o el otro. Desde el Capítulo 6 se sabe que esta recurrencia admite la solución

$$C_N \approx N\lg N.$$

Aunque las cosas no siempre van tan bien, lo que sí que es cierto es que los elementos de partición caen en el centro, *por término medio*. El tener en cuenta la probabilidad exacta de cada posición del elemento de partición complica la relación de recurrencia y la hace más difícil de resolver, pero el resultado final es similar.

Propiedad 9.1 *El Quicksort utiliza del orden de $2N\ln N$ comparaciones por término medio.*

La fórmula exacta de recurrencia para el número de comparaciones utilizadas por la ordenación rápida para una permutación aleatoria de N elementos es

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}), \text{ para } N \geq 2 \text{ con } C_1 = C_0 = 0.$$

El término $N + 1$ cubre el coste de comparar el elemento de partición con cada uno de los otros (más dos extra para el cruce de punteros); el resto viene de la observación de que cada elemento k tiene la probabilidad $1/k$ de ser el elemento de partición, tras lo que quedan archivos aleatorios de tamaño $k - 1$ y $N - k$.

Aunque parece algo complicada, esta recurrencia es realmente fácil de resolver en tres pasos. Primero, $C_0 + C_1 + \dots + C_{N-1}$ es lo mismo que $C_{N-1} + C_{N-2} + \dots + C_0$, por lo que se tiene

$$C_N = N + 1 \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}.$$

Segundo, se puede eliminar el sumatorio multiplicando ambos miembros por N y restando la misma fórmula para $N - 1$:

$$NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}.$$

Esto se simplifica a la recurrencia

$$NC_N = (N+1)C_{N-1} + 2N.$$

Tercero, dividiendo ambos miembros por $N(N+1)$ se obtiene una simplificación en cadena de la recurrencia:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} = \dots = \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}.$$

Esta solución exacta es casi igual a un sumatorio, que se puede aproximar fácilmente por una integral:

$$\frac{C_N}{N+1} \approx 2 \sum_{1 \leq k \leq N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} dx \approx 2\ln N,$$

lo que conduce al resultado esperado. Se observa que $2N\ln N \approx 1,38 N\lg N$, por lo que el número medio de comparaciones es sólo un 38% más alto que el caso más favorable.■

Por tanto, la implementación anterior tiene un comportamiento muy bueno para archivos aleatorios, lo que hace que este método de ordenación sea muy adecuado para muchas aplicaciones. Sin embargo, si se va a utilizar la ordenación un gran número de veces o si se va a aplicar para ordenar un gran archivo, sería útil implementar algunas de las mejoras descritas posteriormente y que hacen menos probable que ocurra un caso negativo, reduciendo el tiempo medio de ejecución en un 20 % y eliminando fácilmente la necesidad de utilizar una clave centinela.

Eliminación de la recursión

Al igual que se hizo en el Capítulo 5, se puede eliminar la recursión del programa del Quicksort utilizando explícitamente una pila donde se imagina que se coloca el «trabajo que queda por hacer» en forma de subarchivos a ordenar. Siempre que se necesite procesar un subarchivo, se sacará de la pila. Al hacer la partición, los dos subarchivos que se crean para procesar se pueden colocar en la pila. Esto conduce a la siguiente implementación no recursiva:

```
void ordenrapido(tipoElemento a[], int izq, int der)
{
    int i; Pila<int> sa(50);
```

```

for (;;)
{
    while (der > izq)
    {
        i = particion(a, izq, der);
        if (i-izq > der-i)
            { sa.meter(1); sa.meter(i-1); izq=i+1; }
        else
            { sa.meter(i+1); sa.meter(der); der=i-1; }
    }
    if (sa.vacia()) break;
    der = sa.sacar(); izq = sa.sacar();
}
}

```

Este programa se diferencia del descrito con anterioridad en dos cuestiones fundamentales. Primero, los dos subarchivos no se colocan en la pila de forma arbitraria, sino que previamente se comprueban sus tamaños y se coloca primero en la pila el más grande de los dos. Segundo, el más pequeño de los dos subarchivos no se coloca en la pila; simplemente se inicializan los valores de los parámetros. Ésta es la técnica de «eliminación de la recursión final» tratada en el Capítulo 5. Para el Quicksort, la combinación de la «eliminación de la recursión final» y la política de procesar primero el más pequeño de los dos subarchivos asegura que la pila necesite espacio solamente para unos $\log N$ elementos, porque cada elemento de la pila, diferente de la cabeza, debe representar a un subarchivo de menos de la mitad de tamaño que el elemento que está debajo de él.

Esto representa un fuerte contraste con el tamaño de la pila en el peor caso de la implementación recursiva, que podría ser tan grande como N (por ejemplo, cuando el archivo ya está ordenado). Ésta es una dificultad sutil pero real de la implementación recursiva del Quicksort: siempre hay una pila subyacente, y un caso degenerado en un gran archivo podría causar una terminación anormal del programa por falta de memoria, comportamiento obviamente indeseable en una rutina de biblioteca de ordenación. Más adelante se verá cómo conseguir que estos casos degenerados sean muy improbables, pero es difícil eliminar este problema en una representación recursiva sin la técnica «eliminación de la recursión final». (Ni siquiera ayuda el invertir el orden en que se procesan los subarchivos.) Por otro lado, algunos compiladores de C++ eliminan automáticamente la recursión y algunas máquinas ofrecen directamente la recursión en el hardware, de manera que en tales entornos el programa anterior podría ser más lento que la implementación recursiva.

La simple utilización de una pila explícita en el programa anterior conduce a programas más seguros y quizás más eficaces que la implementación recursiva directa. Si *los dos* subarchivos tienen sólo un elemento, se coloca en la pila un

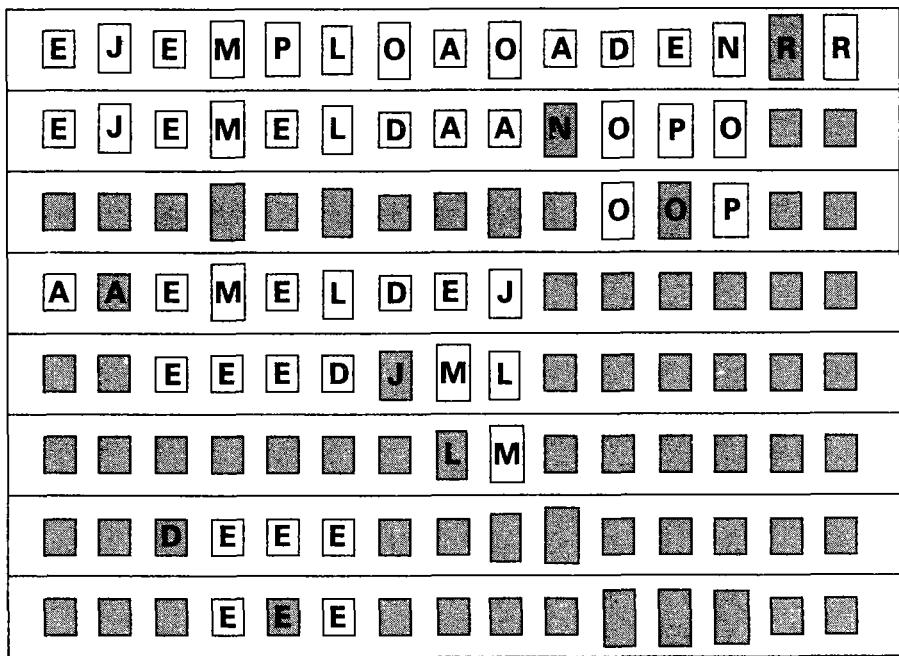


Figura 9.4 Subarchivos en el Quicksort (no recursivo).

subarchivo con der = izq únicamente para ser descartado inmediatamente. Es fácil cambiar el programa para que no coloque tales archivos en la pila. Este cambio es aún más efectivo cuando se incluye la mejora que se describe a continuación, ya que implica ignorar de la misma forma a los subarchivos pequeños, y así serán mucho mayores las posibilidades de que ambos subarchivos no se tengan en cuenta.

Por supuesto, el método no recursivo procesa los mismos subarchivos que el recursivo, para cualquier archivo; simplemente lo hace en orden diferente. La Figura 9.4 muestra las particiones en el caso del ejemplo: las dos primeras particiones son las mismas, pero después el método no recursivo divide primero el subarchivo de la derecha de N porque es más pequeño que el de la izquierda, etcétera.

Si «se unen» las Figuras 9.3 y 9.4 y se conecta cada elemento de partición a

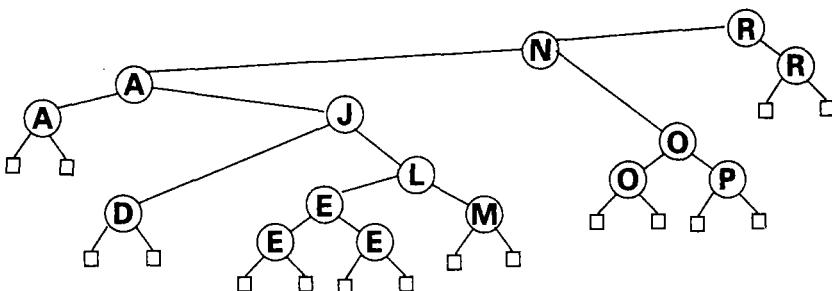


Figura 9.5 Árbol del proceso de partición del Quicksort.

su homólogo de los dos subarchivos, se obtendrá la representación estática del proceso de partición mostrado en la Figura 9.5. En este árbol binario, cada subarchivo se representa por su elemento de partición (o por su único elemento, si es de tamaño uno), y los subárboles de cada nodo son los árboles que representan los subarchivos después de la partición. Los nodos del árbol representados por un cuadrado son los subarchivos nulos. (Para mayor claridad, la segunda A, la D, la M, la O, la P, las dos E finales y la R tienen dos subarchivos nulos: tal y como se vio anteriormente, las variantes del algoritmo tratan de distinta forma los subarchivos nulos.) La implementación recursiva de la ordenación rápida consiste en recorrer los nodos de este árbol por orden previo; la implementación no recursiva se corresponde con la regla de «visitar primero el subárbol más pequeño». En el Capítulo 14 se verá cómo este árbol conduce a una relación directa entre el Quicksort y un método fundamental de búsqueda.

Subarchivos pequeños

La segunda mejora de la ordenación rápida surge al observar que un programa recursivo necesariamente se llama a sí mismo para muchos subarchivos pequeños, por lo que se debería utilizar el mejor método posible cuando se encuentren subarchivos de este tipo. Una forma evidente de hacer esto consiste en modificar la comprobación al principio de la rutina recursiva «*if (der > izq)*» para poder hacer una llamada a la ordenación por inserción (modificada para aceptar los parámetros que definan al subarchivo a ordenar), es decir, «*if (der - izq <= M) insercion(izq, der).*» Aquí *M* es algún parámetro cuyo valor exacto depende de la implementación. El valor elegido para *M* no necesita ser el mejor posible: el algoritmo trabaja casi lo mismo para cualquier *M* con valores entre 5 y 25. La reducción del tiempo de ejecución es del orden de un 20% para la mayoría de las aplicaciones.

Una forma ligeramente más fácil de ordenar subarchivos pequeños, que además es algo más eficaz, consiste tan sólo en cambiar la comprobación del

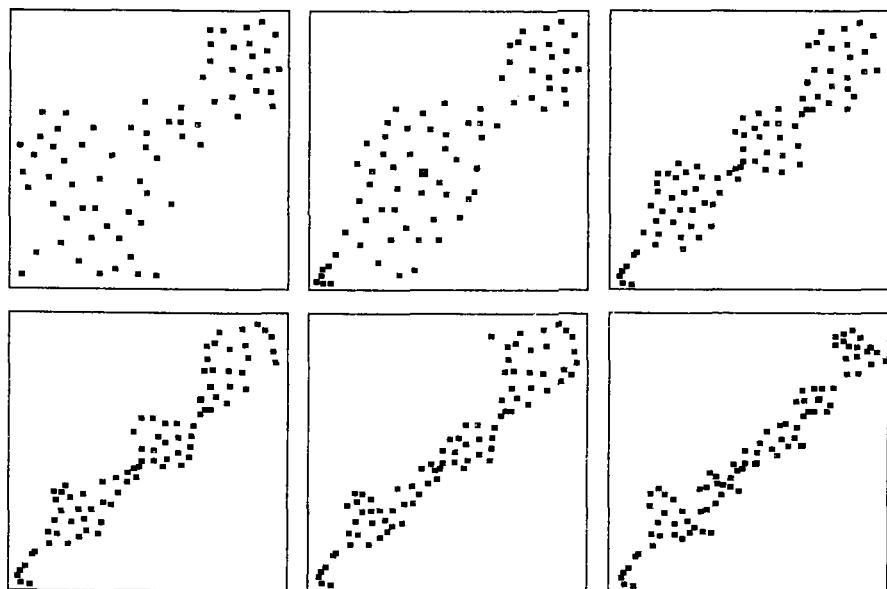


Figura 9.6 Quicksort (recursivo, ignorando los subarchivos pequeños).

principio por «if ($\text{der} - \text{izq} > M$)»: es decir, simplemente ignorar los subarchivos pequeños durante la partición. En la implementación no recursiva se haría esto evitando poner en la pila los archivos menores que M . Tras la partición, lo que se obtiene es un archivo que está casi ordenado. Sin embargo, como se mencionó en el capítulo anterior, la ordenación por inserción es el método a elegir para ordenar tales archivos. Es decir, la ordenación por inserción es tan eficaz para este tipo de archivos como para el conjunto de archivos pequeños que se obtendría si se utilizara directamente. Este método debe emplearse con precaución porque probablemente la ordenación por inserción ordene siempre, incluso si Quicksort tiene un error que impide que funcione. La única evidencia de que algo va mal puede ser el coste excesivo.

La Figura 9.6 da una visión de este proceso en un array grande, ordenado al azar. Estos diagramas representan gráficamente cómo cada partición divide un subarray en dos subproblemas independientes que deberían abordarse por separado. En estas figuras se muestra un subarray en cada uno de los cuadrados, que contiene cuadros de puntos aleatoriamente reordenados; el proceso de partición divide el cuadrado en otros dos más pequeños, con un elemento (el de partición) sobre la diagonal. Los elementos que no están implicados en la partición acabarán bastante cerca de la diagonal y el array resultante se opera fácilmente mediante la ordenación por inserción. Tal y como se indicó antes, el diagrama correspondiente a una implementación no recursiva de Quicksort es similar, pero las particiones se hacen en un orden diferente.

Partición por la mediana de tres

La tercera mejora de la ordenación rápida consiste en utilizar un elemento de partición mejor. Aquí hay varias posibilidades. Para evitar el peor caso, la elección más segura sería utilizar como elemento de partición un elemento aleatorio del array. Entonces, la probabilidad del peor caso será muy baja. Éste es un sencillo ejemplo de un «algoritmo probabilista», que utiliza números pseudoaleatorios para tener casi siempre un buen rendimiento, independientemente del orden de los datos de entrada. Los números pseudoaleatorios pueden ser una herramienta útil en el diseño de algoritmos, en especial si se sospecha alguna tendencia en los datos de entrada. En el caso del Quicksort probablemente es excesivo utilizar un generador de números aleatorios completo sólo con este fin: será suficiente con un número arbitrario (ver Capítulo 35).

Una mejora más útil consiste en tomar tres elementos del archivo y después utilizar la mediana de los tres como elemento de partición. Si los tres elementos elegidos provienen de la izquierda, del centro y de la derecha del array, se puede evitar el uso de centinelas de la siguiente manera: ordenar los tres elementos (utilizando el método de los tres intercambios del capítulo anterior), después intercambiar el del medio con $a[der-1]$, y a continuación, ejecutar el algoritmo de partición sobre $a[iZq+1], \dots, a[der-2]$. A esta mejora se le conoce como el método de la partición por *la mediana de tres*.

Este método mejora el Quicksort de tres formas. En primer lugar, hace mucho más improbable que ocurra el peor caso en cualquier ordenación real. Esto es así porque para que la ordenación dure un tiempo N_2 , dos de los tres elementos examinados deberían estar entre los más grandes o los más pequeños de los elementos del archivo, y esto debería ocurrir en la mayoría de las particiones. En segundo lugar, elimina la necesidad de la clave centinela al hacer la partición porque esta función la realizan los tres elementos examinados antes de la partición. En tercer lugar, de hecho reduce el total del tiempo medio de ejecución del algoritmo en un 5 % aproximadamente.

La combinación de una implementación no recursiva, del método de la partición por la mediana de tres y de un tratamiento aislado de subarchivos pequeños puede mejorar el tiempo de ejecución de la ordenación rápida alrededor de un 25 a 30% con respecto a la implementación recursiva directa. Son posibles otras mejoras algorítmicas (por ejemplo, podría utilizarse la mediana de cinco o más elementos), pero la cantidad de tiempo que se gana es despreciable. Podrían lograrse ahorros de tiempo más significativos (con menos esfuerzo) codificando los bucles internos (o el programa completo) en lenguaje ensamblador o de máquina. Este camino no se recomienda, excepto posiblemente para expertos en aplicaciones de ordenación importantes.

Selección

Una aplicación relacionada con ordenaciones, en las que no siempre es necesaria una ordenación total, es la operación de encontrar la mediana de un conjunto de números. Éste es un cálculo usual en estadística y en diversas aplicaciones de proceso de datos. Una forma de proceder sería ordenar los números y seleccionar el del medio, pero se puede hacer mejor utilizando el proceso de partición de la ordenación rápida.

La operación de encontrar la mediana es un caso particular de la operación de *selección*: encontrar el k -ésimo elemento más pequeño de un conjunto de números. Puesto que ningún algoritmo puede garantizar que un elemento sea el k -ésimo más pequeño sin haber examinado e identificado los $k - 1$ elementos que son menores que él y los $N - k$ elementos que son mayores, la mayoría de los algoritmos de selección pueden devolver todos los k elementos más pequeños de un archivo sin una gran cantidad de cálculos extra.

La selección tiene muchas aplicaciones en el proceso de datos experimentales y de otro tipo. Es muy común el uso de la mediana y de otras *estadísticas de orden* para dividir un archivo en grupos más pequeños. A menudo sólo ha de guardarse para procesos posteriores una pequeña parte de un gran archivo; en tales casos, podría ser más apropiado un programa que pueda seleccionar, por ejemplo, el 10% más significativo de los elementos del archivo, en lugar de otro que hiciera una ordenación total.

Ya se ha visto un algoritmo que puede adaptarse directamente a la selección. Si k es muy pequeño, entonces la *ordenación por selección* será muy eficaz, necesitando un tiempo proporcional a Nk : primero encuentra el elemento más pequeño, después el segundo más pequeño, buscando el más pequeño de los que quedan, etc. Para un k algo más grande, se presentarán en el Capítulo 11 métodos que pueden adaptarse para que se ejecuten en un tiempo proporcional a $N \log k$. Puede formularse un método interesante a partir del procedimiento de partición empleado en el Quicksort, que se ejecute en un tiempo lineal sobre la media de todos los valores de k . Recuérdese que el método de partición de la ordenación rápida reordena un array $a[1], \dots, a[N]$ y devuelve un entero i tal que $a[1], \dots, a[i-1]$ son menores o iguales que $a[i]$ y $a[i+1], \dots, a[N]$ son mayores o iguales que $a[i]$. Si se busca el k -ésimo elemento del archivo y se tiene que $k == i$, entonces ya está hecho. Por el contrario, si $k < i$ se tendrá que buscar el k -ésimo elemento más pequeño en el subarchivo izquierdo y si $k > i$ entonces se tendrá que buscar el $(k-i)$ -ésimo elemento más pequeño del subarchivo derecho. Ajustando esto para encontrar el k -ésimo elemento más pequeño de un array $a[izq], \dots, a[der]$ se llega al siguiente programa:

```
void selecc(tipoElemento a[], int izq, int der, int k)
{
    int i;
```

```

if (der > izq)
{
    i = particion(a, izq, der);
    if (i > izq+k-1) selecc(a, izq, i-1, k);
    if (i < izq+k-1) selecc(a, i+1, der, k-i);
}
}

```

Este procedimiento reordena el array de forma que $a[izq], \dots, a[l-1]$ sean menores o iguales que $a[k]$ y $a[k+1], \dots, a[der]$ sean mayores o iguales que $a[k]$. Por ejemplo, la llamada a `selecc(1, N, (N+1)/2)` divide al array sobre su mediana. Para las claves del ejemplo de ordenación, este programa utiliza sólo cinco llamadas recursivas para encontrar la mediana, como se muestra en la Figura 9.7. Se reordena el archivo para que la mediana esté en un lugar tal que a la izquierda están los elementos más pequeños que ella y los más grandes están a la derecha (los elementos iguales podrían estar en cualquier lado), pero no está totalmente ordenado.

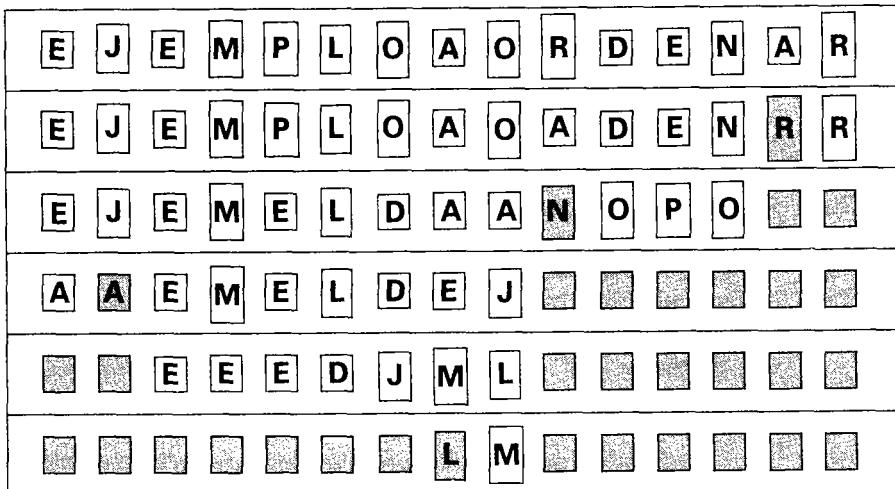


Figura 9.7 Partición para encontrar la mediana.

Puesto que el procedimiento `selecc` siempre termina con una llamada a sí mismo, cuando llegue el momento de la llamada recursiva se podrá simplemente reinicializar los parámetros y volver al principio (no se necesita una pila para eliminar la recursión). También es posible eliminar los cálculos simples que afectan a k , como en la siguiente implementación:

```

void selecc(tipoElemento a[], int N, int k)
{

```

```

int i, j, izq, der; tipoElemento v;
izq = 1; der = N;
while (der > izq)
{
    v = a[der]; i = izq-1; j = der;
    for (;;)
    {
        while (a[++i] < v) ;
        while (a[--j] > v) ;
        if (i >= j) break;
        intercambio(a, i, j);
    }
    intercambio(a, i, der);
    if (i >= k) der = i-1;
    if (i <= k) izq = i+1;
}
}

```

Se emplea un procedimiento de partición idéntico al que se utilizó en la ordenación rápida y, como éste, se podría modificar ligeramente si se esperan muchas claves iguales.

La Figura 9.8 muestra el proceso de selección en un archivo (aleatorio) más grande. Como en el Quicksort se puede afirmar (muy a grandes rasgos) que en un archivo muy grande cada partición debería dividirlo en dos mitades y por tanto todo el proceso necesitaría aproximadamente $N + N/2 + N/4 + N/8 + \dots = 2N$ comparaciones. Al igual que en la ordenación rápida, este argumento aproximado no está muy lejos de la realidad.

Propiedad 9.2 *La selección basada en el Quicksort es de tiempo lineal por término medio.*

Un análisis similar, pero significativamente más complejo, que el dado anteriormente para el Quicksort conduce al resultado de que el número medio de comparaciones es del orden de $2N + 2k\ln(N/k) + 2(N-k)\ln(N/(N - k))$, que es lineal para cualquier valor permitido de k . Para $k = N/2$ (búsqueda de la mediana), resulta aproximadamente $(2 + 2\ln 2)N$ comparaciones.■

El peor caso es muy similar al que se da en la ordenación rápida: utilizar este método para encontrar el elemento más pequeño de un archivo ya ordenado daría como resultado un tiempo de ejecución cuadrático. Podría utilizarse un elemento de partición arbitrario o aleatorio, pero con mucho cuidado: por ejemplo, si se busca el elemento más pequeño, probablemente no se quiera dividir el archivo por la mitad. Es posible modificar el procedimiento de selección basado en la ordenación rápida para garantizar que el tiempo de ejecución sea

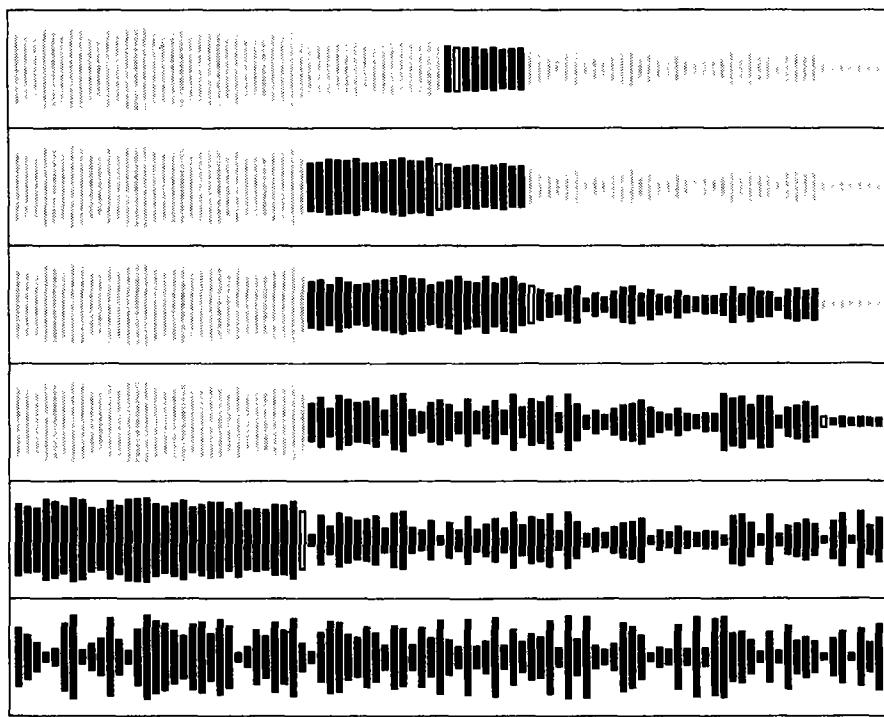


Figura 9.8 Búsqueda de la mediana.

lineal. Estas modificaciones, aunque importantes en teoría, son extremadamente complejas y no del todo prácticas.

Ejercicios

1. Implementar una versión recursiva del Quicksort que ordene por inserción los subarchivos con menos de M elementos, y determinar empíricamente el valor de M para que el método, aplicado a un archivo aleatorio de 1.000 elementos, funcione más rápido.
2. Resolver el problema anterior para una implementación no recursiva.
3. Resolver el problema anterior incorporando la mejora de la mediana de tres.
4. ¿Cuánto tiempo tardará el Quicksort en ordenar un archivo de N elementos iguales?
5. ¿Cuál es el número máximo de veces, durante la ejecución de Quicksort, que puede desplazarse el elemento más grande?

6. Mostrar cómo se divide el archivo A B A B A B A, utilizando los dos métodos sugeridos en el texto.
7. ¿Cuántas comparaciones utiliza el Quicksort para ordenar las letras C U E S T I O N F A C I L?
8. ¿Cuántas claves «centinela» se necesitan si la ordenación por inserción se llama directamente desde el Quicksort?
9. ¿Sería razonable utilizar una cola en lugar de una pila para una implementación no recursiva del Quicksort? ¿Por qué sí o por qué no?
10. Escribir un programa para reordenar un archivo de forma que *todos* los elementos con claves iguales a la mediana estén en su lugar, con los elementos más pequeños a la izquierda y los más grandes a la derecha.

Ordenación por residuos

En muchas aplicaciones de ordenación, las «claves» utilizadas para definir el orden de los registros de los archivos pueden ser muy complicadas. (Considérese, por ejemplo, el orden utilizado en una guía de teléfonos o en el catálogo de una biblioteca.) Debido a esto, es preferible definir los métodos de ordenación en términos de las operaciones básicas de «comparar» dos claves e «intercambiar» dos registros. La mayoría de los métodos que se han estudiado pueden describirse por medio de estas dos operaciones fundamentales. Sin embargo, en muchas aplicaciones se aprovecha el hecho de que las claves pueden considerarse como números de algún intervalo finito. Los métodos de ordenación que aprovechan las propiedades numéricas de estos números se denominan *ordenaciones por residuos*. Estos métodos no sólo comparan las claves: además procesan y comparan fragmentos de ellas.

Los algoritmos de ordenación por residuos tratan las claves como números representados en un sistema de numeración en base M , para diferentes valores de M (el *residuo*), y trabajan con las cifras que forman los números. Por ejemplo, considerando un oficinista que tiene que ordenar un conjunto de tarjetas que tienen impresos números de tres dígitos, una manera razonable de proceder sería hacer diez montones: uno para los números menores de 100, otro para los números que están entre 100 y 199, etc., poner las tarjetas en los montones y, a continuación, tratarlos de forma individual, bien utilizando el mismo método con las cifras siguientes o bien, si sólo quedan unas pocas tarjetas, utilizando algún método más sencillo. Éste es un simple ejemplo de una ordenación por residuos para $M = 10$. En este capítulo se estudiará con detalle éste y algún otro método. Por supuesto, para la mayoría de las computadoras es más apropiado trabajar con $M = 2$ (o alguna potencia de 2) que con $M = 10$.

Todo lo que esté representado dentro de una computadora digital puede tratarse como un número binario, por lo que muchas aplicaciones de ordenación pueden volverse a escribir para hacer factible la utilización de la ordenación por residuos que operan con claves que sean números binarios. Por fortuna, C++ proporciona operadores de bajo nivel que hacen posible implementar tales ope-

raciones de una manera directa y eficaz. Esto es importante porque hay otros muchos lenguajes (como por ejemplo Pascal) que intencionadamente hacen que sea difícil escribir un programa que dependa de la representación binaria de los números.

Bits

Dado (una clave representada como) un número binario, la operación fundamental necesaria para la ordenación por residuos es la extracción de un conjunto contiguo de bits del número. Si, por ejemplo, se van a procesar claves que son números enteros comprendidos entre 0 y 1.000, se puede suponer que éstos valores están representados por números binarios de 10 bits. En lenguaje de máquina, los bits se extraen de los números binarios utilizando operaciones de manipulación de bits, tales como la «y» y los desplazamientos. Por ejemplo, los dos bits más significativos de un número de 10 bits se extraen desplazándolos ocho posiciones a la derecha y haciendo a continuación una operación binaria «y» con la máscara 0000000011. En C++, estas operaciones se realizan directamente con los operadores de manipulación de bits `>>` y `&`. Por ejemplo, los dos bits más significativos de un número `x` de 10 bits se obtienen por `(x >> 8) & 03`. En general, «poner a cero todos los bits de `x` excepto los `j` que están más a la derecha» se puede obtener con `x & ~(~0<<j)` porque `~(~0<<j)` es una máscara con unos en las `j` posiciones de bits de la derecha y con ceros en el resto.

Hasta el momento, en las implementaciones de los algoritmos de ordenación se ha dejado sin especificar el tipo de las claves de los elementos que se van a ordenar (`tipoElemento`), con la suposición implícita de que los operadores de comparación `<`, `==`, y `>` estarán disponibles para claves usuales tales como números enteros, números de coma flotante, cadenas de caracteres. En los algoritmos de ordenación por residuos no se utilizan los operadores de comparación, pero C++ permite ser explícitos respecto a los operadores que *se deseé* utilizar, tal y como se muestra en la siguiente definición para claves de enteros:

```
class clavebits
{
private:
    int x;
public:
    clavebits& operator=(int i)
    { x = i; return *this; }
    inline unsigned bits (int k, int j)
    { return (x >>k) & ~(~0<<j); }
};

typedef clavebits tipoElemento;
```

Esto significa que sólo se utilizarán dos operaciones en las claves tipo-Elemento: asignación de un valor entero y bits. El código de asignación es un estándar de C++. El operador bits utiliza las instrucciones de tratamiento de bits descritas anteriormente para devolver los j bits de x que están a la derecha de k bits. Por ejemplo, si t es del tipo tipoElemento entonces la declaración $t = 1000$ asigna simplemente 1.000 (en binario 1111101000) al campo de datos de t , entonces $t.bits(2, 4)$ es 2 (10 en binario, los bits quinto y sexto de la derecha de t). Para utilizar los algoritmos de ordenación por residuos se debe tener un operador bits, de manera que las claves a ordenar puedan aparecer de forma lógica como cadenas de bits. Se pueden incluir otras cosas en el tipo, tales como el número máximo de bits de una clave o una forma de permitir que cadenas de bits de longitud variable puedan ser claves. Como siempre, se omitirán tales detalles para poder centrarse en las propiedades esenciales de los algoritmos.

Armados con esta herramienta básica, se estudiarán dos tipos de ordenación por residuos que se diferencian en el orden en el que se examinan los bits de las claves. Se supone que las claves no son cortas, de manera que merece la pena hacer el esfuerzo de extraer sus bits. Si las claves son cortas, entonces se puede utilizar el método de cuenta de distribuciones del Capítulo 8. Recuérdese que este método puede ordenar N claves enteras entre 0 y $M - 1$ en un tiempo lineal, utilizando una tabla auxiliar de tamaño M para los contadores y otra de tamaño N para los registros reordenados. De este modo, si se puede disponer de una tabla de tamaño 2^b , se puede ordenar fácilmente claves de una longitud de b bits en un tiempo lineal. La ordenación por residuos es útil si las claves son suficientemente grandes (a partir de $b = 32$), donde no es posible esta ordenación por cuenta de distribuciones.

El primer método básico de ordenación por residuos que se estudiará se denomina *ordenación por intercambio de residuos* y examina los bits de las claves de izquierda a derecha, manipulando los registros de una forma similar a la ordenación rápida. El segundo método, denominado *ordenación directa por residuos*, examina los bits de las claves de derecha a izquierda y se puede ejecutar en tiempo lineal en una serie razonable de circunstancias.

Ordenación por intercambio de residuos

Supóngase que se pueden reordenar los registros de un archivo de manera que todas aquellas claves que comiencen con el bit 0 se coloquen delante de todas las que comiencen con el bit 1. Esto define un método de ordenación recursivo como el de la ordenación rápida: si se ordenan los dos subarchivos independientemente, todo el archivo estará ordenado. Para reorganizar el archivo se examina éste empezando por la izquierda para encontrar una clave que empiece con el bit 1 y empezando por la derecha para encontrar una clave que

empiece con el bit 0, intercambiándolas y continuando así hasta que se crucen los punteros:

```
void cambioresiduos(tipoElemento a[], int izq, int der, int b)
{
    int i, j; tipoElemento t;
    if (der>izq, && b>=0)
    {
        i = izq; j = der;
        while (j != i)
        {
            while (!a[i].bits(b, 1) && i<j) i++;
            while (a[j].bits(b, 1) && j>i) j--;
            intercambio(a, i, j)
        }
        if (!a[der].bits(b, 1)) j++;
        cambioresiduos(a, izq, j-1, b-1);
        cambioresiduos(a, j, der, b-1);
    }
}
```

La llamada a `cambioresiduos(1, N, 30)` ordenará el array si `a[1], ..., a[N]` son enteros positivos menores que 2^{32} (de manera que se puedan representar como números binarios de 31 bits). La variable `b` «guarda la pista» del bit que se está examinando, variando entre 30 (el que está más a la izquierda) y 0 (el que está más a la derecha). El número exacto de bits a utilizar depende de una manera directa de la aplicación, del número de bits por palabra de la máquina y de la representación de los números enteros y de los negativos.

Evidentemente esta implementación es bastante similar a la implementación recursiva del método de ordenación rápida del Capítulo 9. En esencia, hacer una partición en el método de ordenación por intercambio de residuos es como hacerlo en el de ordenación rápida, excepto que en lugar de utilizar como elemento de partición un número del archivo aquí se utiliza el número 2^b . Como se podría dar el caso de que 2^b no pertenezca al archivo, no está garantizado que durante la partición se coloque todo elemento en su lugar definitivo. Además, como sólo se examina un bit, no se puede contar con centinelas para detener al puntero durante la exploración, por lo que se incluyen las comprobaciones ($i < j$) en los bucles de exploración. Esto se traduce en un intercambio extra para el caso ($i == j$), que podría evitarse con una instrucción `break`, como en la implementación de la ordenación rápida; aunque en este caso el «intercambio» de `a[i]` consigo mismo no tiene consecuencias. El proceso de partición se detendrá cuando `j` sea igual a `i` y todos los elementos a la derecha de `a[i]` tengan bits 1 en la posición b -ésima y todos los elementos a la izquierda de `a[i]` ten-

gan bits 0 en la posición b -ésima. El mismo elemento $a[i]$ tendrá un bit 1 *a menos que* todas las claves del archivo tengan un 0 en la posición b . La implementación anterior tiene una comprobación extra justo después del bucle de partición, para cubrir este caso.

La Figura 10.1 muestra cómo el ejemplo de archivo de claves se divide y ordena por este método. Se puede comparar con la Figura 9.2 de la ordenación rápida, aunque la operación del método de partición es completamente opaca sin la representación binaria de las claves. La Figura 10.2 muestra la partición en términos de la representación binaria de las claves. Se utiliza un código sencillo de cinco bits, presentando a la letra i -ésima del alfabeto mediante la representación binaria del número i . Esto es una versión simplificada de la codificación real de caracteres, que utiliza más bits (siete u ocho) y representa a más caracteres (mayúsculas, minúsculas, números, símbolos especiales). Traduciendo las claves de la Figura 10.1 a este código de caracteres de cinco bits, comprimiendo la tabla de manera que el subarchivo dividido se represente «en paralelo», y no uno por línea, y transponiendo después filas y columnas, se puede mostrar en la Figura 10.2 cómo los bits más significativos de las claves controlan la partición. En esta figura cada partición se indica en el siguiente diagrama a la derecha por un subarchivo «0» blanco seguido por un subarchivo «1» gris, excepción hecha de los subarchivos de tamaño 1 que desaparecen del proceso de partición en cuanto se los encuentra.

Un serio problema potencial de la ordenación por residuos es que con frecuencia se pueden dar particiones degeneradas (particiones en las que todas las claves tienen el mismo valor que el bit que se está utilizando). Esta situación surge frecuentemente en archivos reales al ordenar números pequeños (con muchos ceros en cabeza). Esto también puede ocurrir para caracteres: por ejemplo, suponiendo que claves de 32 bits están formadas por grupos de cuatro caracteres, codificados en el código estándar de ocho bits, entonces probablemente se den particiones degeneradas en las primeras posiciones de cada carácter, ya que, por ejemplo, todas las minúsculas comienzan con los mismos bits en la mayoría de los códigos de caracteres. Hay otros muchos efectos similares que deben tenerse en cuenta al ordenar datos codificados.

En la Figura 10.2 se puede ver que una vez que se distingue una clave del resto de ellas por sus bits izquierdos, no se vuelve a examinar ningún otro bit. Esto es una ventaja en algunas ocasiones y un inconveniente en otras. La ventaja se da cuando los bits de las claves son verdaderamente aleatorios, ya que en este caso cada clave difiere de las otras en unos $\lg N$ bits, lo que podría ser mucho menor que el número de bits de las claves. Esto es así porque, en una situación aleatoria, se espera que cada partición divida el subarchivo por la mitad. Por ejemplo, ordenar un archivo con 1.000 registros podría traer consigo el examinar aproximadamente 10 u 11 bits de cada clave (incluso cuando las claves son de 32 bits). Por el contrario, hay que destacar que se examinan todos los bits de las claves iguales; la ordenación por residuos no funciona bien en archivos que contienen muchas claves iguales. La ordenación por intercambio de residuos es de hecho algo más rápida que la ordenación rápida si las claves que se

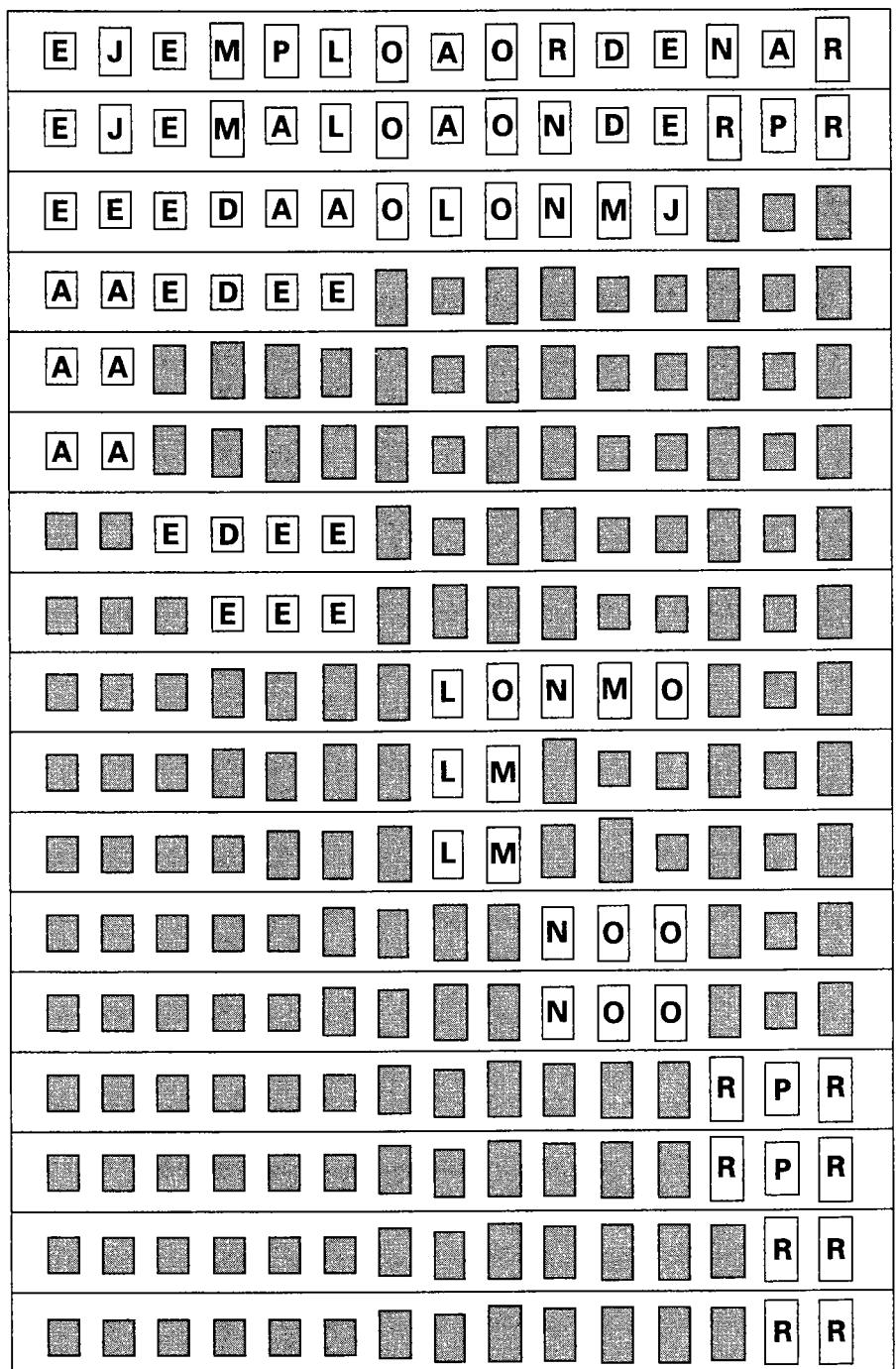


Figura 10.1 Subarchivos en la ordenación por intercambio de residuos.

Figura 10.2 Ordenación por intercambio de residuos (ordenación por residuos «de izquierda a derecha»).

quiere ordenar están formadas por bits verdaderamente aleatorios, mientras que la ordenación rápida se adapta mejor a las situaciones menos aleatorias.

En la Figura 10.3 se puede ver el árbol que representa el proceso de partición de la ordenación por intercambio de residuos, que se puede comparar con la Figura 9.5. En este árbol binario, los nodos internos representan a los puntos de partición, y los nodos externos son las claves del archivo que terminan todas en subarchivos de tamaño 1. En el Capítulo 17 se verá cómo este árbol sugiere una relación directa entre la ordenación por intercambio de residuos y un método fundamental de búsqueda.

La implementación recursiva anterior se puede mejorar eliminando la recursión y tratando de forma diferente a los subarchivos pequeños, tal y como se hizo en la ordenación rápida.

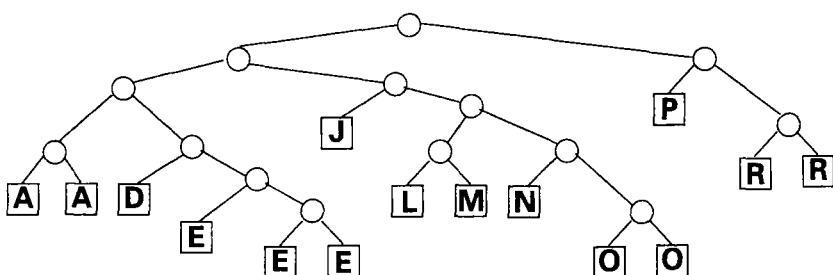


Figura 10.3 Diagrama de árbol del proceso de partición en una ordenación por intercambio de residuos.

E 00101	J 01010	P 10000	P 10000	P 10000	A 00001
J 01010	P 10000	L 01100	A 00001	A 00001	A 00001
E 00101	L 01100	D 00100	A 00001	A 00001	D 00100
M 01101	R 10010	E 00101	J 01010	R 10010	E 00101
P 10000	D 00100	E 00101	R 10010	R 10010	E 00101
L 01100	N 01110	M 01101	R 10010	D 00100	E 00101
O 01111	R 10010	A 00001	L 01100	E 00101	J 01010
A 00001	E 00101	E 00101	D 00100	E 00101	L 01100
O 01111	E 00101	A 00001	E 00101	E 00101	M 01101
R 10010	M 01101	J 01010	E 00101	J 01010	N 01110
D 00100	O 01111	R 10010	M 01101	L 01100	O 01111
E 00101	A 00001	N 01110	E 00101	M 01101	O 01111
N 01110	O 01111	R 10010	V 01110	N 01110	O 10000
A 00001	E 00101	O 01111	O 01111	O 01111	R 10010
R 10010	A 00001	O 01111	O 01111	O 01111	R 10010

Figura 10.4 Ordenación directa por residuos (ordenación por residuos «de derecha a izquierda»).

Ordenación directa por residuos

Un método alternativo de la ordenación por residuos consiste en examinar los bits de derecha a izquierda. Éste era el método utilizado por las antiguas máquinas encargadas de ordenar las tarjetas perforadas de las computadoras: un mazo de tarjetas se procesaba 80 veces, una por cada columna, procediendo de derecha a izquierda. La Figura 10.4 muestra cómo funciona una ordenación por residuos de derecha a izquierda, bit a bit, sobre el ejemplo de archivo de claves. En los diagramas, la i -ésima columna se ordena mediante el rastreo de los bits que están en la posición i de la clave y se deduce de la columna anterior ($(i - 1)$ -ésima) extrayendo todas las claves que tienen un 0 en el bit i -ésimo y después todas las que tienen un 1 en el i -ésimo bit.

No es fácil convencerse de que el método funciona; de hecho no lo hace a menos que el proceso de partición sobre un bit sea estable. Una vez que se ha identificado la importancia de la estabilidad, se puede encontrar una prueba trivial de que sí funciona: después de poner las claves que tienen un 0 en el i -ésimo bit, delante de las que tienen un 1 en el i -ésimo bit (de una manera estable), se sabe que dadas dos claves cualesquiera están en el orden adecuado (conforme a los bits ya examinados) en el archivo, bien porque sus i -ésimos bits son diferentes, en cuyo caso la partición los coloca en el orden adecuado, o bien porque son iguales, en cuyo caso ya están en el orden correcto debido a la estabilidad. El requisito de estabilidad significa, por ejemplo, que el método de partición utilizado en la ordenación por intercambio de residuos no se puede utilizar en esta ordenación de derecha a izquierda.

El proceso de una partición es parecido a ordenar un archivo con sólo dos

valores y que la ordenación por cuenta de distribuciones es muy apropiada para esto. Si se supone que $M = 2$ en el programa de la cuenta de distribuciones y se reemplaza $a[i]$ por $\text{bits}(a[i], k, 1)$, entonces el programa se convierte en un método para ordenar los elementos del array a tomando los bits que están en la posición k empezando por la derecha y colocando el resultado en el array temporal b. Pero no existe ninguna razón para utilizar $M = 2$; de hecho, se debería hacer M tan grande como sea posible, puesto que se necesita una tabla de M contadores. Esto corresponde a utilizar m bits a la vez durante la ordenación, con $M = 2^m$. Así, la ordenación directa por residuos se convierte en poco más que en una generalización de la ordenación por cuenta de distribuciones, como puede verse en la siguiente implementación que permite ordenar $a[1], \dots, a[N]$ para los w bits más a la derecha:

```
void directaresiduos(tipoElemento a[], tipoElemento b[], int N)
{
    int i, j, pasar, contador[M-1];
    for (pasar = 0; pasar < w/m; pasar++)
    {
        for (j = 0; j < M; j++) contador[j] = 0;
        for (i = 1; i <= N; i++)
            contador[a[i].bits(pasar*m, m)]++;
        for (j = 1; j < M; j++)
            contador[j] += contador[j-1];
        for (i = N; i >= 1; i--)
            b[contador[a[i].bits(pasar*m, m)]--] = a[i];
        for (i = 1; i <= N; i++) a[i] = b[i];
    }
}
```

Esta implementación supone que el procedimiento de llamada pasa el array auxiliar como un parámetro de entrada al mismo tiempo que el array a ordenar. La correspondencia $M = 2^m$ se ha preservado en los nombres de las variables, pero los lectores deben tener en cuenta que en algunos entornos de programación no podrán establecer la diferencia entre m y M .

El procedimiento anterior funciona adecuadamente sólo si w es múltiplo de m . Por lo regular, esto no será una restricción que haya que asumir para la ordenación por residuos: tan sólo corresponde a dividir las claves a ordenar en un número entero de partes del mismo tamaño. Cuando $m=w$ se obtiene la ordenación por cuenta de distribuciones; cuando $m=1$ se obtiene la ordenación directa por residuos, la ordenación por residuos de derecha a izquierda y bit a bit, descrita en el ejemplo anterior.

La implementación anterior mueve el archivo de a hasta b durante cada fase de la cuenta de distribuciones, después vuelve a a con un sencillo bucle. Este

bucle de «copia de array» podría eliminarse, si se desea, haciendo dos copias del código de dicha cuenta, una para ordenar de a hacia b y la otra para ordenar de b hacia a.

Características de rendimiento de la ordenación por residuos

Los tiempos de ejecución de las dos ordenaciones por residuos básicas, para ordenar N registros con claves de b bits, son esencialmente Nb . Por un lado se puede pensar que este tiempo de ejecución es equivalente a $N\log N$, ya que si los números son todos diferentes, b debe ser al menos $\log N$. Por otro lado, los dos métodos realizan normalmente muchas menos de Nb operaciones: el método de izquierda a derecha, porque se puede detener en cuanto se hayan encontrado las diferencias entre claves, y el método de derecha a izquierda, porque puede procesar muchos bits a la vez.

Propiedad 10.1 *La ordenación por intercambio de residuos examina, por término medio, aproximadamente $N\lg N$ bits.*

Si el tamaño del archivo es una potencia de dos y los bits son aleatorios, se puede esperar que la mitad de los bits más significativos sean 0 y la otra mitad sean 1, por lo que la recurrencia $C_N = 2C_{N/2} + N$ podría describir el rendimiento, como en el caso de la ordenación rápida del Capítulo 9. De nuevo, esta descripción de la situación no es exacta, porque el elemento de partición solamente coincide con el centro en el caso medio (y porque las claves tienen un número finito de bits). Sin embargo, en este modelo, es mucho más probable que dicho elemento esté en el centro que en la ordenación rápida, de manera que la propiedad resulta ser cierta. (Para probar esto se necesita un análisis detallado que está más allá del alcance de este libro.)■

Propiedad 10.2 *Las dos ordenaciones por residuos examinan menos de Nb bits para ordenar N claves de b bits.*

En otras palabras, la ordenación por residuos es *lineal* en el sentido de que el tiempo que se necesita es proporcional al número de bits de la entrada. Esto se deduce directamente estudiando los programas: ningún bit se examina más de una vez.■

Para archivos aleatorios grandes, la ordenación por intercambio de residuos tiene un comportamiento parecido a la ordenación rápida, como se muestra en la Figura 9.6; pero la ordenación directa por residuos se comporta de forma muy diferente. La Figura 10.5 muestra las etapas de la ordenación directa por residuos de un archivo con claves de cinco bits. En estos diagramas aparece claramente la organización progresiva del archivo durante la ordenación. Por ejemplo, después de la tercera etapa (abajo a la izquierda), el archivo está formado

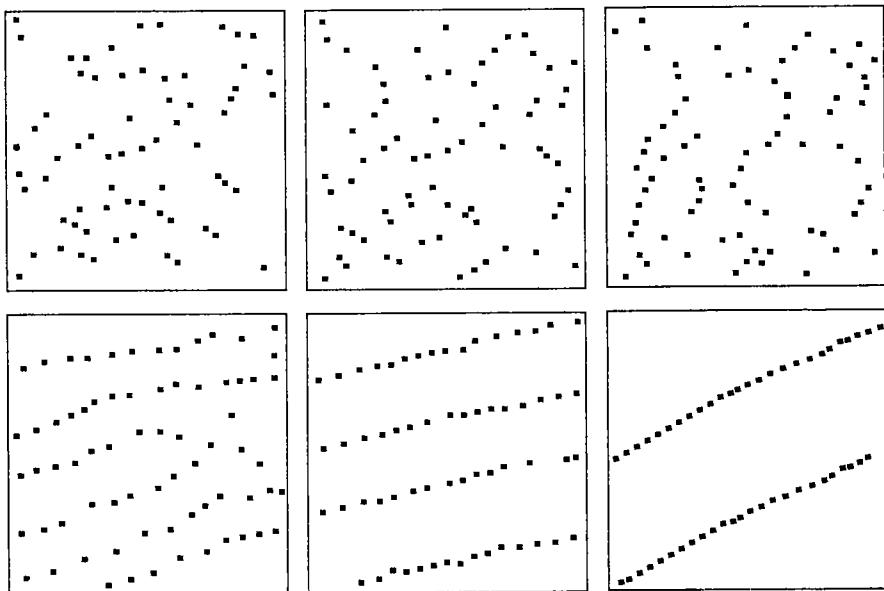


Figura 10.5 Etapas de una ordenación directa por residuos.

por cuatro subarchivos entremezclados: las claves que comienzan por 00 (banda inferior), las claves que comienzan por 01, etcétera.

Propiedad 10.3 *La ordenación directa por residuos puede ordenar N registros con claves de b bits en b/m pasos, utilizando un espacio extra de 2^m contadores (y una memoria intermedia para reorganizar el archivo).*

La demostración de esta propiedad se deduce directamente de la implementación. En particular, si se puede tomar $m = b/4$ sin utilizar demasiada memoria extra, se obtiene una ordenación lineal. Las consecuencias prácticas de esta propiedad se tratarán con más detalle a continuación. ■

Una ordenación lineal

La implementación de la ordenación directa por residuos, dada en la sección anterior, efectúa b/m pasos a través del archivo. Haciendo m muy grande se obtiene un método de ordenación muy eficaz, al menos mientras se disponga de $M = 2^m$ palabras de memoria disponible. Una elección razonable consiste en tomar m con un valor aproximado a la cuarta parte del tamaño de una palabra

($b/4$), ya que de esta manera la ordenación por residuos se hará en cuatro pasadas de la cuenta de distribuciones. Se tratan las claves como números en base M y se examina cada dígito (en base M) de cada clave, aunque sólo hay cuatro dígitos por clave. (Esto se corresponde directamente con la organización de la arquitectura de muchas computadoras: una organización representativa tiene palabras de 32 bits, cada una de ellas formada por cuatro bytes de ocho bits. En este caso, el procedimiento bits acaba extrayendo ciertos bytes de las palabras, lo que se puede realizar fácilmente en tales computadoras.) Ahora, cada pasada de la cuenta de distribuciones es lineal y, como sólo hay cuatro, toda la ordenación es lineal. Por consiguiente, éste es el mejor rendimiento que se podría esperar de una ordenación.

De hecho, incluso podría bastar con sólo dos pasadas de la cuenta de distribuciones. (A estas alturas es probable que incluso un lector cuidadoso tenga dificultades para distinguir derecha de izquierda, por lo que puede ser necesario hacer un pequeño esfuerzo para comprender este método.) Se realiza aprovechando el hecho de que si sólo se utilizan los $b/2$ bits más significativos de las claves de b bits, el archivo está *casi* ordenado. Al igual que se hizo en la ordenación rápida, se puede completar eficazmente la ordenación, aplicando más tarde la ordenación por inserción sobre la totalidad del archivo. Este método es una modificación trivial de la implementación anterior: para hacer una ordenación de derecha a izquierda utilizando la mitad delantera de las claves, simplemente se comienza el bucle exterior con $\text{pasar} = b/(2*m)$ en lugar de empezar con $\text{pasar} = 0$. A continuación se puede aplicar una ordenación por inserción convencional al archivo casi ordenado que se obtiene. Para convenirse de que un archivo ordenado en sus bits más significativos está bastante bien ordenado, el lector puede examinar las primeras columnas de la Figura 10.2. Por ejemplo, aplicar la ordenación por inserción sobre un archivo ya ordenado en sus tres primeros bits necesitaría solamente seis intercambios.

Utilizando dos pasadas de la cuenta de distribuciones (siendo m aproximadamente igual a la cuarta parte del tamaño de una palabra) y utilizando después una ordenación por inserción para terminar el trabajo, se obtiene un método de ordenación que probablemente se ejecutará más rápido que cualquiera de los que se han visto para grandes archivos cuyas claves son bits aleatorios. El principal inconveniente es que se necesita un array auxiliar del mismo tamaño que el que se está ordenando. Este array extra se puede eliminar utilizando técnicas de listas enlazadas; pero aun así todavía se necesita un espacio extra proporcional a N (para los enlaces).

Es obvio que en muchas aplicaciones lo deseable es una ordenación lineal, pero existen razones por las que no es la panacea que se podría imaginar. En primer lugar, su eficacia depende mucho de que las claves estén formadas por bits aleatorios y que estén ordenadas aleatoriamente. Si no se cumplen estas condiciones es de esperar un rendimiento muy degradado. En segundo lugar, se necesita un espacio extra proporcional al tamaño del array que se está ordenando. En tercer lugar, el «bucle interno» del programa contiene bastantes instrucciones, así que, aun siendo lineal, no será mucho más rápido que, por ejem-

plo, el de ordenación rápida, como cabría esperar, excepto para archivos bastante grandes (pero en éstos el array extra se convierte en un verdadero obstáculo).

La ordenación por residuos podría caracterizarse como una aproximación a una ordenación de «utilidad especial», porque su viabilidad depende de propiedades especiales de las claves, en contraste con la «utilidad general» de algoritmos tales como el de ordenación rápida, que se utilizan mucho más porque se adaptan a una gran variedad de aplicaciones. La elección entre la ordenación rápida y la ordenación por residuos depende no solamente de las características de la aplicación (como la clave, el registro y el tamaño del archivo), sino también de las características del entorno de programación y de la máquina, que están íntimamente relacionadas con la eficacia de acceso y la manipulación individual de los bits. En aplicaciones adecuadas, la ordenación por residuos se puede ejecutar dos veces más rápidamente que la ordenación rápida, o incluso más; pero podría no merecer la pena si el espacio es un problema potencial o si las claves son de tamaño variable o no son necesariamente aleatorias, o ambas cosas.

Ejercicios

1. Comparar el número de intercambios efectuados por la ordenación por intercambio de residuos y la de ordenación rápida para el archivo 001, 011, 101, 110, 000, 001, 001, 010, 111, 110, 010.
2. ¿Por qué no es tan importante eliminar la recursión de la ordenación por intercambio de residuos como lo era para la ordenación rápida?
3. Modificar el programa de ordenación por intercambio de residuos para ignorar los bits más significativos que son idénticos en todas las claves. ¿En qué situaciones sería ventajosa esta técnica?
4. Verdadero o falso: el tiempo de ejecución de la ordenación directa por residuos no depende del orden de las claves en el archivo de entrada. Razonar la respuesta.
5. ¿Qué método es probablemente más rápido para un archivo con todas las claves iguales: el de ordenación por intercambio de residuos o el de ordenación directa por residuos?
6. Verdadero o falso: tanto la ordenación por intercambio de residuos como la ordenación directa por residuos examinan todos los bits de todas las claves del archivo. Razonar la respuesta.
7. Aparte del requisito de memoria extra, ¿cuál es el mayor inconveniente de la estrategia de realizar la ordenación directa por residuos sobre los bits más significativos de las claves y terminar después con una ordenación por inserción?
8. ¿Cuánta memoria se necesita exactamente para hacer una ordenación directa por residuos, en cuatro pasadas, de N claves de b bits?

9. ¿Qué tipo de archivo de entrada hará que la ordenación por intercambio de residuos se ejecute lo más lentamente posible (para un N muy grande)?
10. Comparar empíricamente la ordenación directa por residuos con la ordenación por intercambio de residuos para un archivo aleatorio de 10.000 claves de 32 bits.

Colas de prioridad

En muchas aplicaciones los registros con claves se deben procesar en orden, pero no necesariamente en orden completo, ni todos a la vez. A veces se forma un conjunto de registros y se procesa el mayor; a continuación posiblemente se incluyan otros elementos y luego se procesa el nuevo registro máximo y así sucesivamente. Una estructura de datos apropiada para un entorno como éste es aquella que permita insertar un nuevo elemento y eliminar el mayor. Esta estructura, que se puede contrastar con las colas (donde se elimina el más antiguo) o con las pilas (donde se elimina el más reciente), se denomina *cola de prioridad*. De hecho, una cola de prioridad se puede considerar como una generalización de las pilas y de las colas (y de otras estructuras de datos simples), puesto que estas estructuras se pueden implementar con colas de prioridad, haciendo las asignaciones de prioridad adecuadas.

Las aplicaciones de las colas de prioridad incluyen sistemas de simulación (donde las claves pueden corresponder a «cronologías de sucesos» que se deben procesar en orden), planificación de tareas en los sistemas informáticos (donde las claves pueden corresponder a «prioridades» que indican qué usuarios se deben procesar en primer lugar) y cálculos numéricos (donde las claves pueden ser errores de cálculo y por tanto se puede tratar en primer lugar el más grande).

Más adelante, en este libro, se verá cómo se pueden utilizar las colas de prioridad como bloques básicos para la construcción de algoritmos más avanzados. En el Capítulo 22 se desarrollará un algoritmo de compresión de ficheros utilizando las rutinas de este capítulo, y en los Capítulos 31 y 33 se verá cómo las colas de prioridad pueden servir de base a muchos algoritmos fundamentales de búsqueda en grafos. Éstos son sólo unos pocos ejemplos del importante papel que desempeñan las colas de prioridad como herramienta básica en el diseño de algoritmos.

Por razones de utilidad se debe precisar algo más sobre la forma de tratar las colas de prioridad, puesto que existen varias operaciones que puede ser necesario llevar a cabo sobre ellas, para preservarlas y poderlas utilizar con eficacia en aplicaciones como las mencionadas anteriormente. En verdad, la razón princi-

pal por la que las colas de prioridad son tan útiles es la flexibilidad con que permiten llevar a cabo eficazmente una gran variedad de operaciones sobre conjuntos de registros con claves. Lo que se desea es construir y mantener una estructura de datos que contenga registros con claves numéricas (*prioridades*) y que cuente con algunas de las operaciones siguientes:

Construir una cola de prioridad a partir de N elementos.

Insertar un nuevo elemento.

Suprimir el elemento más grande.

Sustituir el elemento más grande por un nuevo elemento (a menos que éste sea mayor).

Cambiar la prioridad de un elemento.

Eliminar un elemento arbitrario determinado.

Unir dos colas de prioridad en una más grande.

(Si los registros pueden tener claves iguales, se considera que el «más grande» significa «uno cualquiera de los registros que tiene el valor de clave más grande».)

La operación *sustituir* es casi equivalente a *insertar* seguida de *suprimir* (la diferencia es que *insertar/suprimir* requiere que la cola de prioridad crezca temporalmente en un elemento); obsérvese que esta operación es diferente de *suprimir* seguido de *insertar*. Ésta se incluye como una operación por separado porque, como se verá, algunas implementaciones de las colas de prioridad pueden realizar eficazmente la operación *sustituir*. Del mismo modo, la operación *cambiar* podría implementarse como *eliminar* seguido de *insertar* y la operación *construir* con el uso repetido de *insertar*, pero estas operaciones se pueden implementar más eficazmente y de manera directa por medio de ciertas estructuras de datos. La operación *unión* necesita estructuras de datos avanzadas; en su lugar se estudiará una estructura de datos «clásica», denominada *montículo*, con la que es posible lograr implementaciones eficaces de las cinco primeras operaciones.

La cola de prioridad, tal como se ha descrito anteriormente, es un excelente ejemplo de la *estructura de datos abstracta*, descrita en el Capítulo 3: está muy bien definida en términos de las operaciones que se llevan a cabo sobre ella, independientemente de cómo se organizan los datos y se procesan en una implementación particular.

Cada implementación diferente de las colas de prioridad se acompaña de diferentes características de rendimiento para las diversas operaciones que se llevan a cabo, lo que conduce a comparaciones de costes. En efecto, las diferencias de rendimiento son realmente las únicas que pueden aparecer en el concepto de estructura de datos abstracta. Se ilustrará este punto presentando algunas estructuras de datos elementales que permiten implementar colas de prioridad. Después se examinará una estructura de datos más avanzada y se mostrará cómo se pueden implementar algunas de las operaciones anteriores utilizando esta estructura. Para concluir se verá un importante algoritmo de ordenación que se obtiene de forma natural a partir de estas implementaciones.

Implementaciones elementales

Una forma de organizar una cola de prioridad es como una *lista no ordenada*, colocando simplemente los elementos en un array $a[1], \dots, a[N]$ sin prestar atención a los valores de las claves. (Como es habitual, se reserva $a[0]$ y $a[N+1]$ para valores centinelas, en el caso de que se necesiten.) El array a y su tamaño N se utilizan solamente por las funciones de la cola de prioridad y se suponen «ocultos» de las rutinas de invocación. Si se usa un array para implementar una lista no ordenada, la clase cola de prioridad se obtiene fácilmente como sigue:

```
class CP
{
    private:
        tipoElemento *a;
        int N;
    public:
        CP(int max)
        { a = new tipoElemento[max]; N = 0; }
        ~CP()
        { delete a; }
        void insertar(tipoElemento v)
        { a[++N] = v; }
        tipoElemento suprimir()
        {
            int j, max = 1;
            for (j = 2; j <= N; j++)
                if (a[j] > a[max]) max = j;
            intercambio(a, max, N);
            return a[N--];
        }
};
```

Para *insertar*, se incrementa N y se coloca el nuevo elemento en $a[N]$, una operación en tiempo constante. Pero *suprimir* requiere recorrer el array para encontrar el elemento con la clave más grande, lo que lleva un tiempo lineal (se deben examinar todos los elementos del array), y después intercambia $a[N]$ con dicho elemento y decremente N . La implementación de *sustituir* es muy similar y por tanto se omite.

Para implementar la operación *cambiar* (cambiar la prioridad del elemento $a[k]$), es suficiente con almacenar el nuevo valor, y para *eliminar* el elemento $a[k]$, se puede cambiar por $a[N]$ y decrementar N , como en la última línea de *suprimir*. Tales operaciones, que hacen referencia a elementos específicos, sólo

tienen sentido en una implementación «indirecta» o con «puntero», donde cada elemento mantiene una referencia a su lugar en la estructura de datos. Una implementación de este tipo se presenta al final de este capítulo.

Otra organización elemental a utilizar es una *lista ordenada*, empleando otra vez un array $a[1], \dots, a[N]$, pero manteniendo los elementos en orden creciente de sus claves. Ahora *suprimir* implica simplemente devolver $a[N]$ y decrementar N (operación de tiempo constante), pero *insertar* necesita desplazar todos los elementos superiores una posición a la derecha, lo que podría llevar un tiempo lineal, y *construir* implicaría una ordenación.

Cualquier algoritmo de cola de prioridad se puede convertir en un algoritmo de ordenación utilizando repetidamente *insertar* para construir una cola de prioridad que contenga todos los elementos a ordenar, y utilizar después repetidamente *suprimir* para vaciar esta última cola y obtener elementos en orden inverso. La utilización de una lista no ordenada para representar de esta forma a una cola de prioridad corresponde a una ordenación por selección; la de una lista ordenada corresponde a una ordenación por inserción.

También se pueden utilizar listas enlazadas como listas no ordenadas o listas ordenadas en lugar de la implementación por array anterior. Esto no cambia las características fundamentales de rendimiento de *insertar*, *suprimir* o *sustituir*, pero permite ejecutar *eliminar* y *unir* en un tiempo constante. Aquí se omiten estas operaciones porque son similares a la lista de operaciones básicas del Capítulo 3 y porque en el Capítulo 14 se dan implementaciones de métodos similares para el problema de búsqueda (encontrar un registro con una clave dada).

Como siempre, es conveniente no olvidarse de estas implementaciones porque a menudo, en muchas situaciones prácticas, superan el rendimiento de métodos más complicados. Por ejemplo, la implementación por lista no ordenada puede ser apropiada en una aplicación donde sólo se llevan a cabo algunas operaciones de «suprimir el más grande» frente a un gran número de inserciones, mientras que una lista ordenada podría ser lo apropiado si los elementos que se insertan en la cola de prioridad siempre tienden a estar próximos al elemento mayor.

Estructura de datos montículo

La estructura de datos que se utilizará para implementar las colas de prioridad implica almacenar los registros en un array de tal manera que se garantice que cada clave es mayor que otras dos que están situadas en posiciones específicas. A su vez, cada una de esas claves debe ser mayor que otras dos y así sucesivamente. Este ordenamiento es muy fácil de representar si se dibuja el array como una estructura de árbol bidimensional, con líneas que descienden desde cada clave hacia las dos que se sabe que son inferiores, como en la Figura 11.1.

Se vio en el Capítulo 4 que esta estructura se denomina «árbol binario com-

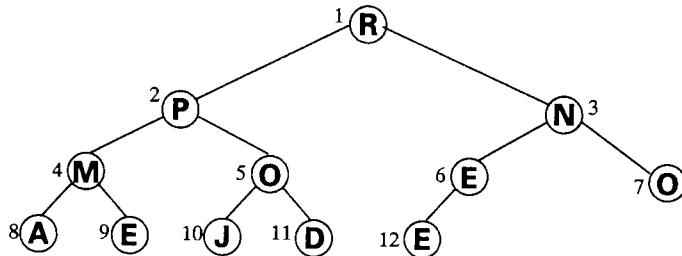


Figura 11.1 Representación de un montículo por un árbol completo.

pleto»: se puede construir colocando un nodo (llamado *raíz*) y luego actuando hacia abajo de la página y de izquierda a derecha, conectando cada par de nodos al del nivel superior bajo el que se encuentran, y así sucesivamente hasta que se hayan colocado N nodos. Los dos nodos debajo de cada nodo se denominan sus *hijos*; el nodo superior de cada nodo se denomina el *padre*. Las claves del árbol deben satisfacer la *condición del montículo*: la clave de cada nodo debe ser superior (o igual) a las claves de sus hijos (si tiene alguno). Esto implica que la clave más grande está en la raíz.

Se pueden representar árboles binarios secuencialmente en un array con sólo poner la raíz en la posición 1, sus hijos en las posiciones 2 y 3, los nodos del nivel inferior en las posiciones 4, 5, 6 y 7, etc., como indican los números de la Figura 11.1. Por ejemplo, la representación por array del árbol anterior se muestra en la Figura 11.2.

Esta representación natural es útil porque es muy fácil ir de un nodo a su padre o a un hijo. El padre del nodo de la posición j está en la posición $j/2$ (redondeado al entero más cercano si j es impar) e, inversamente, los dos hijos del nodo j están en las posiciones $2j$ y $2j + 1$. Esto hace que recorrer este árbol sea más fácil que si estuviera implementado por la representación enlazada estándar (en la que cada elemento contiene punteros a su padre y a sus hijos). La estructura rígida de árboles binarios completos representados por arrays limita su utilidad como estructura de datos, pero tiene la flexibilidad suficiente para permitir la implementación de los algoritmos de cola de prioridad. Un *montículo* es un árbol binario completo representado por un array, en el cual cada nodo satisface la condición del montículo. En particular, la clave más grande está siempre en la primera posición del array.

Todos los algoritmos operan a lo largo de algún *camino* desde la raíz hasta

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	R	P	N	M	O	E	O	A	E	J	D	E

Figura 11.2 Representación de un montículo por un array.

el fondo del montículo (moviéndose del padre a un hijo o de un hijo al padre). Es fácil observar que en un montículo de N nodos, todos los caminos tienen $\lg N$ nodos. (Hay alrededor de $N/2$ hijos en el fondo, $N/4$ nodos cuyos hijos son los del fondo, $N/8$ nodos con nietos en el fondo, etc. Cada «generación» tiene alrededor de la mitad de nodos que la siguiente, lo que implica que puede haber como máximo $\lg N$ generaciones.) Por tanto, todas las operaciones de colas de prioridad (excepto *unir*) se pueden hacer en tiempos logarítmicos, utilizando un montículo.

Algoritmos sobre montículos

Todos los algoritmos de colas de prioridad que trabajan con montículos comienzan haciendo una simple modificación estructural, que podría violar la condición del montículo, y luego lo recorren, modificándolo, para asegurar que dicha condición se satisfaga en todos los nodos. Algunos algoritmos recorren el montículo de abajo hacia arriba, otros lo hacen desde arriba hacia abajo. En todos los algoritmos se supondrá que los registros son enteros de una palabra almacenados en un array *a* de un cierto tamaño máximo y que el entero *N* indica el tamaño actual del montículo. Como antes, se supone que el array y su tamaño son accesibles solamente para las rutinas de las colas de prioridad: los datos se pasan entre el usuario y la cola únicamente a través de las llamadas a las rutinas.

Para poder construir un montículo, primero es necesario implementar la operación *insertar*. Puesto que esta operación incrementa el tamaño del montículo en uno, se debe incrementar *N*. A continuación se coloca el registro a insertar en *a[N]*, lo que puede violar la condición del montículo, en cuyo caso (el nuevo nodo es mayor que el padre) se intercambia el nuevo nodo con su padre. Esto puede, a su vez, causar una violación que se debe arreglar de la misma forma. Por ejemplo, si se va a insertar *R* en el montículo de la Figura 11.1, primero se almacena en *a[N]* como el hijo derecho de *E*. Luego, puesto que es más grande que *E*, se intercambia con este nodo *y*, puesto que es más grande que *N*, se intercambia con él, y el proceso termina dado que es igual que *R*. Se obtiene como resultado el montículo que se muestra en la Figura 11.3.

El código para este método es directo: la implementación siguiente utiliza *subirmonticulo(N)* para eliminar la violación de la condición, después de insertar un nuevo elemento en *N*:

```
void CP::subirmonticulo(int k)
{
    tipoElemento v;
    v = a[k]; a[0] = elementoMAX;
    while (a[k/2] <= v)
```

```

    { = a[k] = a[k/s]; k = k/2; }
    a[k] = v;
}
void CP::insertar(tipoElemento v)
{a[++N] = v; subirmonticulo(N); }

```

Si se reemplazase $k/2$ por $k-1$ en todo el programa anterior, se tendría, en esencia, un paso de la ordenación por inserción (implementando una cola de prioridad por medio de una lista ordenada); en lugar de ello, aquí se está insertando la nueva clave a lo largo del camino desde N a la raíz. Al igual que con la ordenación por inserción, no es necesario hacer un intercambio completo en el bucle, porque v siempre está implicado en estos intercambios. Se debe poner una clave centinela en $a[0]$ para detener el bucle en el caso en que v sea mayor que todas las claves del montículo. Más adelante se verá otro empleo de $a[0]$.

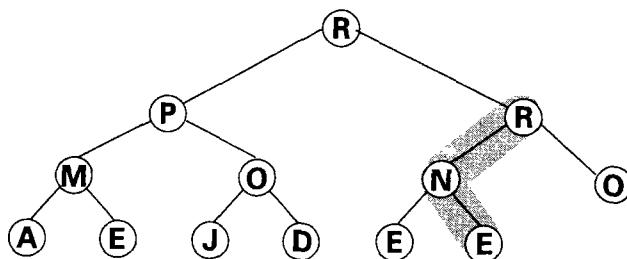


Figura 11.3 Inserción de un nuevo elemento (R) en un montículo.

La operación sustituir consiste en poner una nueva clave en la raíz y luego moverse hacia abajo por el montículo, desde la cima hacia el fondo, para restaurar la condición del mismo. Por ejemplo, si la R del montículo anterior se reemplaza por C, el primer paso es poner a C en la raíz. Esto viola la condición del montículo, pero se puede arreglar intercambiando C con R, el mayor de los dos hijos de la raíz. Esto provoca otra violación en el nivel siguiente, la cual se puede de nuevo arreglar intercambiando C con el mayor de los dos hijos (O en este caso). El proceso continúa hasta que no se viole la condición del montículo en el nodo ocupado por C. En el ejemplo, C desciende hasta el penúltimo nivel quedando el montículo que se muestra en la Figura 11.4.

La operación «suprimir el mayor» implica casi el mismo proceso. Puesto que el montículo tendrá un elemento menos después de la operación, es necesario decrementar N, desalojando el elemento que estaba almacenado en la última posición. Pero, como se va a suprimir el elemento más grande (que está en $a[1]$), la operación suprimir consiste en sustituir, utilizando el elemento que estaba en $a[N]$. El montículo de la Figura 11.5 es el resultado de suprimir R del montículo de la Figura 11.4 reemplazándolo por E y moviéndose a continua-

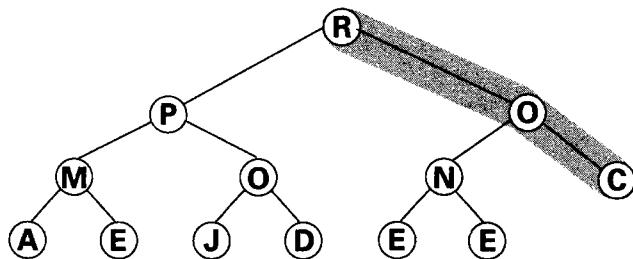


Figura 11.4 Sustitución (por C) de la clave más grande del montículo.

ción hacia abajo promocionando al mayor de los dos hijos, hasta que se alcance un nodo con sus dos hijos inferiores a E, lo que en este caso lleva al fondo del montículo.

La implementación de ambas operaciones se basa en la técnica de ir restaurando hacia arriba un montículo para ir satisfaciendo la condición del montículo en todos los lugares, excepto posiblemente en la raíz. Si la clave de la raíz es muy pequeña, se debe mover hacia abajo por el montículo sin violar la condición en ninguno de los nodos que se tocan. Esto indica que se puede utilizar la misma operación para restaurar el montículo después de disminuir el valor de una posición cualquiera. Esto se puede implementar como sigue:

```

void CP::bajarmonticulo(int k)
{
    int j; tipoElemento v;
    v = a[k];
    while (k <= N/2)
    {
        j = k+k;
        if (j<N && a[j]<a[j+1]) j++;
        if (v >= a[j]) break;
        a[k] = a[j];
        k = j;
    }
    a[k] = v;
}
  
```

Este procedimiento recorre hacia abajo el montículo, intercambiando el nodo de la posición k con el mayor de sus dos hijos, si es necesario, y parando cuando el nodo k sea mayor que sus dos hijos o se haya alcanzado el fondo. (Como es posible que el nodo k tenga un solo hijo, ¡este caso se debe tratar de manera adecuada!) Como antes, no se necesita efectuar un intercambio completo porque v siempre está implicado en cada uno de ellos. El bucle interno de este pro-

grama es un ejemplo de un bucle con dos salidas distintas: una para el caso en que se alcance el fondo del montículo (como en el primer ejemplo anterior), y otra para el caso en que la condición del montículo se satisfaga en algún lugar del interior del mismo. Éste es el prototipo de las situaciones que necesitan una instrucción break.

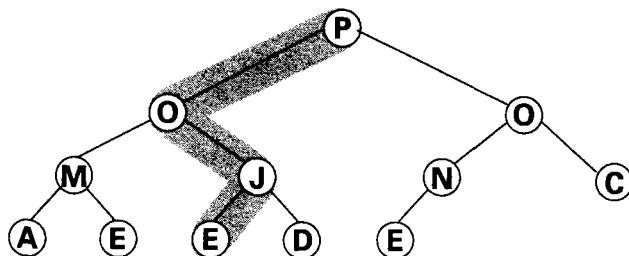


Figura 11.5 Supresión del elemento mayor de un montículo.

Ahora la operación *suprimir* es una aplicación directa de este procedimiento:

```

tipoElemento CP::suprimir()
{
    tipoElemento v = a[1];
    a[1] = a[N--];
    bajarmonticulo(1);
    return v;
}
  
```

El valor devuelto es el que se encuentra inicialmente en $a[1]$. Después el elemento en $a[N]$ se pone en $a[1]$ y se decrementa el tamaño del montículo, quedando solamente por hacer una llamada a *bajarmonticulo* para restaurar la condición del montículo en todas partes. La operación *sustituir* es un tanto más complicada:

```

tipoElemento CP::sustituir(tipoElemento v)
{
    a[0] = v;
    bajarmonticulo(0);
    return a[0];
}
  
```

Este código utiliza $a[0]$ de una forma artificial: sus hijos son el propio 0 y 1, por tanto si v es mayor que el elemento más grande del montículo, el montículo no se toca; en caso contrario, se coloca v en el montículo y se devuelve $a[1]$.

La operación *extraer* un elemento arbitrario del montículo y la operación *cambiar* también se pueden implementar utilizando una combinación sencilla de los métodos anteriores. Por ejemplo, si se aumenta la prioridad del elemento de la posición k , entonces se debe llamar a *subirmonticulo*, y si se disminuye entonces la tarea la hace *bajarmonticulo*.

Propiedad 11.1 *Todas las operaciones básicas —insertar, suprimir, sustituir, (bajarmonticulo, subirmonticulo), eliminar y cambiar— necesitan menos de $2\lg N$ comparaciones cuando se llevan a cabo sobre un montículo de N elementos.*

Todas estas operaciones implican recorrer un camino entre la raíz y el fondo del montículo, que no puede incluir más de $\lg N$ elementos en un montículo de tamaño N . El factor dos proviene de *bajarmonticulo*, que hace dos comparaciones en su bucle interno; las otras operaciones necesitan sólo $\lg N$ comparaciones. ■

Es importante señalar que la operación *unir* no se ha incluido en esta lista. Realizar eficazmente esta operación necesita una estructura de datos mucho más sofisticada. A pesar de todo, en muchas aplicaciones, se podría esperar que esta operación se solicite con mucha menos frecuencia que las otras.

Ordenación por montículos

Las operaciones básicas sobre montículos estudiadas con anterioridad permiten definir un método elegante y eficaz de ordenación. Dicho método, denominado *ordenación por montículos*, no utiliza memoria extra y garantiza ordenar N elementos en alrededor de $N\lg N$ pasos, sin importar cuál sea la entrada. Por desgracia, su bucle interno es algo más largo que el del *Quicksort*, y, como promedio, es dos veces más lento.

La idea es simplemente construir un montículo que contenga los elementos a ordenar y después suprimirlos todos en orden. Una forma de ordenar consiste en insertar los elementos en un montículo vacío, como en las dos primeras líneas del código siguiente (que de hecho sólo implementa *construir(a,N)*), y después efectuar N operaciones *suprimir*, colocando el elemento que se ha suprimido en el lugar que ha quedado vacante en el montículo que se está comprimiendo:

```
void ordenmonticulo(tipoElemento a[], int N)
{
    int i; CP monticulo(N);
    for (i = 1; i <= N; i++) monticulo.insertar(a[i]);
    for (i = N; i >= 1; i--) a[i] = monticulo.suprimir();
}
```

Los procedimientos de implementación de colas de prioridad se han utilizado

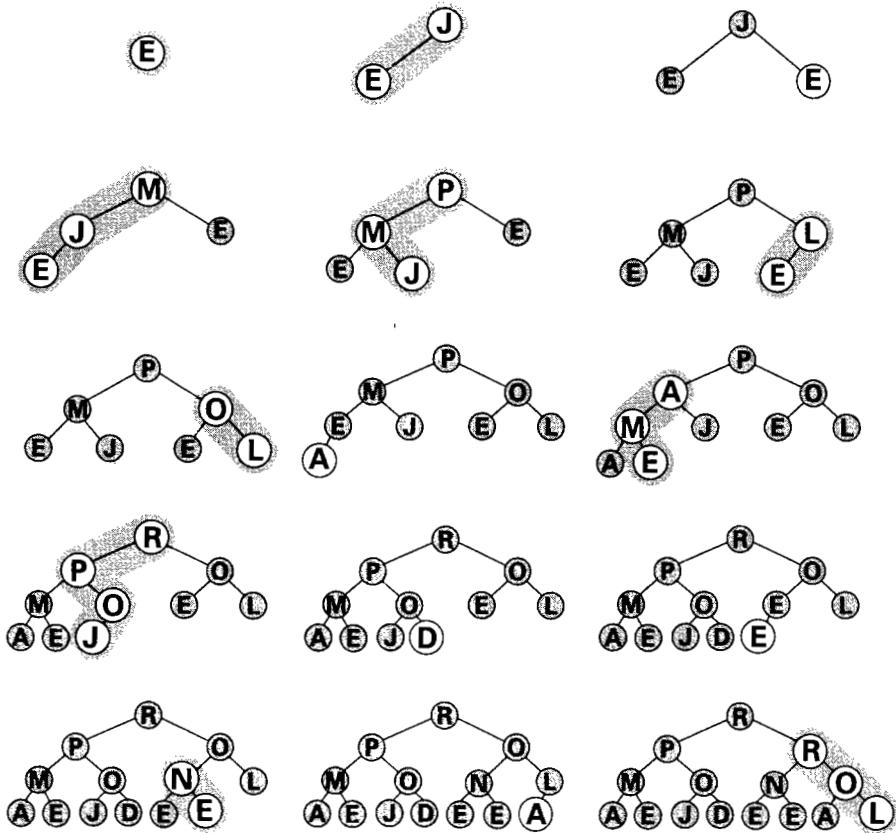


Figura 11.6 Construcción descendente (de arriba hacia abajo) del montículo.

sólo con propósitos descriptivos: en una implementación real de la ordenación, sería más simple utilizar el código de los procedimientos para evitar llamadas innecesarias a los mismos. Y lo que es más importante, el programa se puede arreglar para que la ordenación se lleve a cabo *in situ* (sin utilizar memoria extra para el montículo), permitiendo que `ordenmonticulo` tenga acceso directo al array y dejando que la cola de prioridad resida en $a[1], \dots, a[k - 1]$.

La Figura 11.6 muestra la construcción del montículo cuando se insertan las claves E J E M P L O A O R D E N A R, en este orden, en un montículo inicialmente vacío, y la Figura 11.7 muestra cómo se ordenan dichas claves quitando la R, luego la otra R, etcétera.

En realidad es mejor construir el montículo retrocediendo a través de él e ir creando pequeños submontículos desde el fondo hacia arriba, como se muestra en la Figura 11.8. Este método considera cada posición del array como la raíz de un pequeño submontículo y se aprovecha del hecho de que `bajarmonti-`

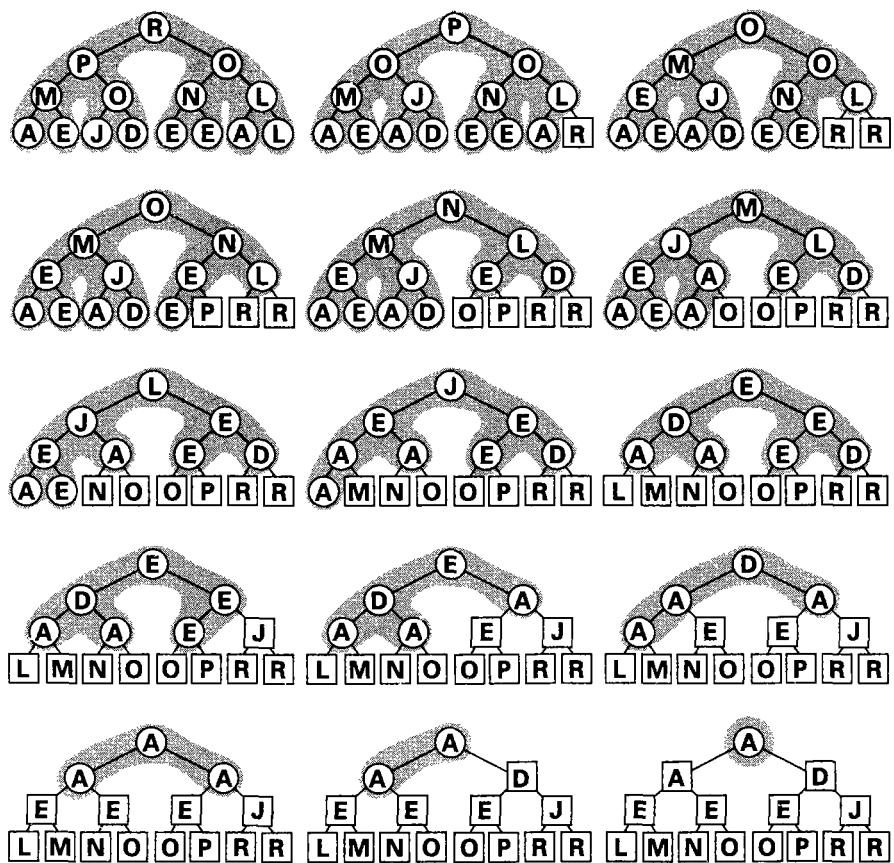


Figura 11.7 Ordenación a partir de un montículo.

culo puede aplicarse en esos submontículos tan bien como en el montículo grande. Trabajando hacia atrás a través del montículo, cada nodo es la raíz de un submontículo que responde a la condición del montículo, excepto posiblemente en su raíz; el método `bajarmonticulo` termina el trabajo. El recorrido comienza en la mitad del camino del array porque los submontículos de tamaño 1 se pueden saltar.

Ya se ha señalado que la operación *suprimir* se puede implementar intercambiando los elementos primero y último, decrementando N, y llamando a `bajarmonticulo(1)`. Esto conduce a la siguiente implementación de la *ordenación por montículos*:

```
void ordenmonticulo(tipoElemento a[], int N)
{
```

```

int k;
for (k = N/2; k >= 1; k--)
    bajarmonticulo(a, N, k);
while (N > 1)
    { intercambio(a, 1, N); bajarmonticulo(a, --N, 1); }
}

```

Aquí otra vez se abandona cualquier idea de ocultar la representación del montículo, y se supone que se ha modificado `bajarmonticulo` para que sus dos primeros argumentos sean el array y el tamaño del montículo. El primer bucle `for` podría utilizarse para implementar un constructor que ordene en tiempo lineal un array en forma de montículo. A continuación, el bucle `while` intercambia el elemento mayor con el último y restaura el montículo, al igual que antes. Es interesante notar que, aunque los bucles de este programa parecen hacer cosas muy diferentes, están construidos tomando en cuenta el mismo procedimiento fundamental.

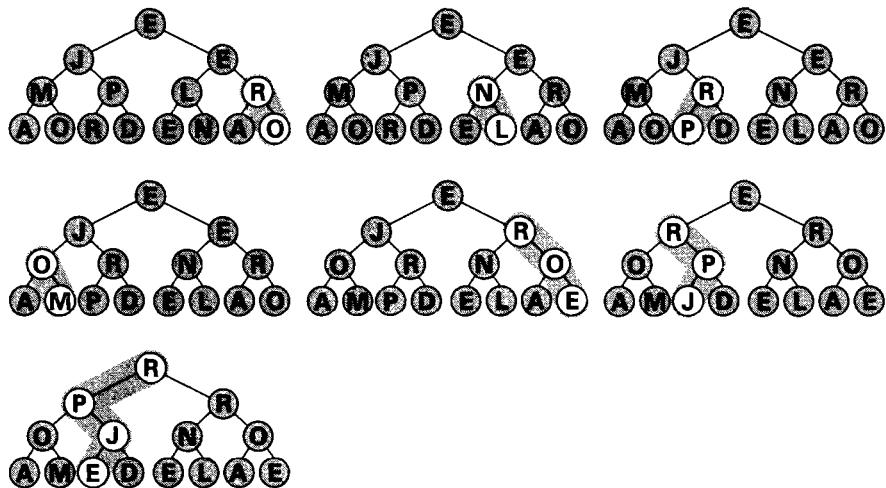


Figura 11.8 Construcción ascendente (de abajo hacia arriba) del montículo.

La Figura 11.9 ilustra el movimiento de los datos en la ordenación por montículos al mostrar el contenido de cada montículo operado por `bajarmonticulo` en el ejemplo de ordenación, justamente después de que `bajarmonticulo` haya hecho que la condición del montículo se cumpla en todas partes.

Propiedad 11.2 *La construcción ascendente (de abajo hacia arriba) del montículo es lineal.*

Esto se deduce del hecho de que la mayoría de los montículos procesados son pequeños. Por ejemplo, para construir un montículo de 127 elementos, el mé-

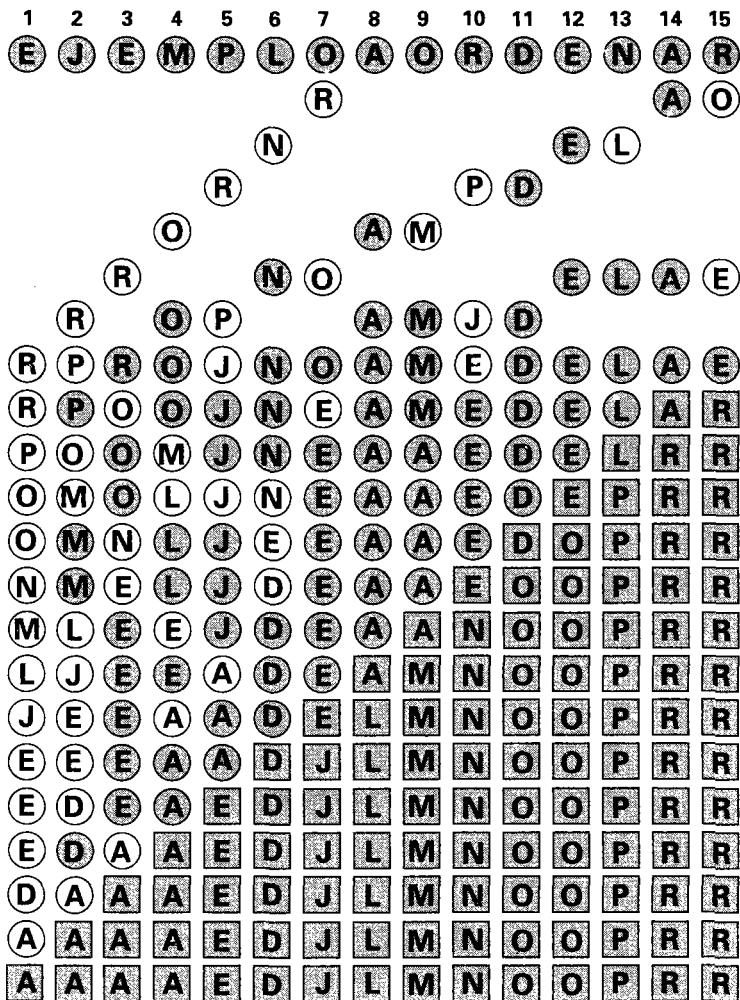


Figura 11.9 Movimiento de datos en la ordenación por montículos.

todo llama a `bajarmonticulo` para 64 montículos de tamaño 1, 32 de tamaño 3, 16 de tamaño 7, 8 de tamaño 15, 4 de tamaño 31, 2 de tamaño 63 y uno de tamaño 127, por tanto se necesitan en el peor caso $64 \cdot 0 + 32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ «promociones» (dos veces más que comparaciones). Para $N = 2^n$, una cota superior del número de comparaciones es

$$\sum_{1 \leq k \leq n} (k-1)2^{n-k} = 2^n - n - 1 < N,$$

y una demostración similar es válida cuando N no es una potencia de dos.■

Esta propiedad no es de particular importancia en la ordenación por montículos, puesto que su tiempo está dominado fundamentalmente por el tiempo $N\log N$ de la ordenación, pero es importante en otras aplicaciones de colas de prioridad, en las que un tiempo lineal de construir puede conducir a un algoritmo lineal. Se observa que construir un montículo con N insertar sucesivos necesita $N\log N$ pasos en el peor caso (aunque, por término medio, tiende a ser lineal).

Propiedad 11.3 *La ordenación por montículos utiliza menos de $2N\lg N$ comparaciones para ordenar N elementos.*

Una cota ligeramente superior, por ejemplo $3N\lg N$, es inmediata a partir de la propiedad 11.1. La cota que se da aquí se obtiene de un cálculo más cuidadoso basado en la propiedad 11.2.■

Como se mencionó anteriormente, la propiedad 11.3 es la razón principal por la que la ordenación por montículos tiene un interés práctico: el número de pasos necesarios para ordenar N elementos es *obligatoriamente* proporcional a $N\log N$, sin importar cuál sea la entrada. A diferencia de los restantes métodos de ordenación que se han visto, no hay ningún «peor caso» que pueda hacer que la ordenación por montículos se ejecute con más lentitud.

Las Figuras 11.10 y 11.11 muestran cómo la ordenación por montículos opera sobre un fichero ordenado aleatoriamente. En la Figura 11.10, el proceso parece cualquier cosa menos una ordenación, puesto que los elementos grandes se desplazan hacia el comienzo del archivo. Pero la Figura 11.11 muestra esta estructura a medida que se ordena el fichero al ir seleccionando los elementos más grandes.

Montículos indirectos

En muchas aplicaciones de las colas de prioridad, no se desea que los registros se estén desplazando continuamente. En lugar de ello, se quiere que las rutinas de las colas no devuelvan valores sino que indiquen *cuál* de los registros es el más grande, etc. Esto es semejante a la «ordenación indirecta» o a la «ordenación por punteros» descrita en el Capítulo 8. Puede ser útil examinar esta técnica con detalle, ya que es muy práctico utilizar los montículos de esta forma.

Una aproximación consiste, como en el Capítulo 8, en hacer de modo que en lugar de estar reorganizando las claves en un array a las rutinas de las colas de prioridad trabajen con un array de índices o de punteros dentro del array, refiriéndose a las claves indirectamente. Más aún, para implementar las operaciones *cambiar* y *eliminar* es necesario conservar la posición de cada elemento del montículo. Esto se puede llevar a cabo con modificaciones cuidadosas del código anterior, de forma similar a la presentada en el Capítulo 8.

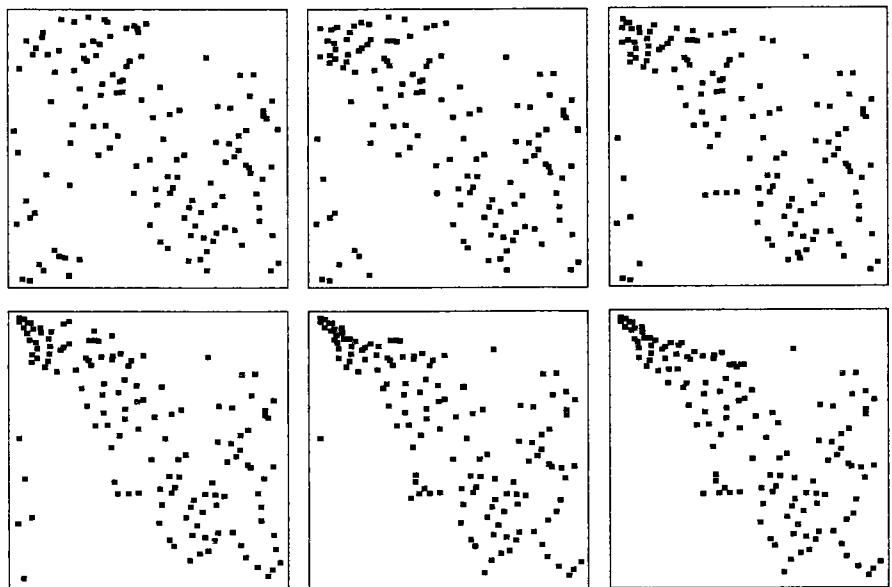


Figura 11.10. Ordenación por montículos de una permutación aleatoria: fase de construcción.

Se adoptará una aproximación, toscamente equivalente, pero un tanto más general, que proporcionará rutinas de colas de prioridad que serán útiles más adelante, en particular en los Capítulos 22 y 31. Se generaliza la interfaz para incluir otro argumento para insertar y cambiar: un entero entre 1 y talla que se asocia con la clave que se inserta en la cola de prioridad. En particular, la operación suprimir es para devolver el entero asociado con la clave más pe-

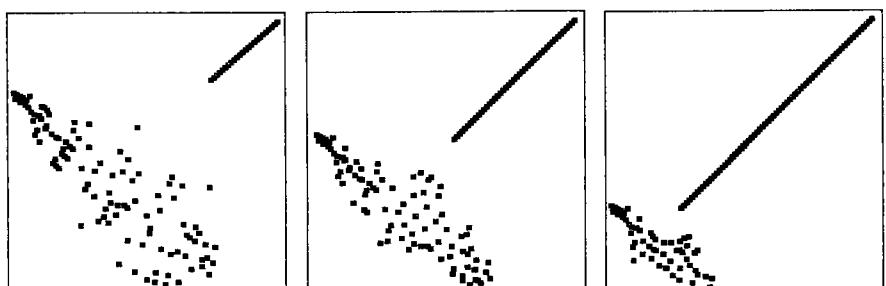


Figura 11.11. Ordenación por montículos de una permutación aleatoria: fase de ordenación.

queña de la cola, no la propia clave. Por ejemplo, si las claves están almacenadas en un array (o si son parte de los registros almacenados en un array), se utilizarán los índices de array con este objetivo. Luego se puede recuperar del array la clave y cualquier otra información asociada.

Para ilustrar cómo se puede implementar esto, considérese una versión modificada de la implementación de «array no ordenado» dada al comienzo del capítulo. Se añade un array info para «seguir el rastro» de la información asociada y un array p para hacerlo con las posiciones de las claves en la cola. Hay que asegurarse de que estos arrays satisfagan que $p[info[x]] = info[p[x]] = x$ y $a[p[x]] = v$ para cada entero x que se haya asociado con alguna clave v de la cola:

```
class CP
{
private:
    int *a, *p, *info;
    int N;
public:
    CP(int talla)
    { a = new tipoElemento[talla];
      p = new tipoElemento[talla];
      info = new int[talla]; N = 0;
    }
    ~CP()
    { delete a; delete p; delete info; }
    void insertar(int x, tipoElemento v)
    { a[++N] = v; p[x] = N; info[N] = x; }
    void cambiar(int x, tipoElemento v)
    { a[p[x]] = v; }
    int suprimir() //suprimir el menor
    {
        int j, min = 1;
        for (j = 2; j <= N; j++)
            if (a[j] < a[min]) min = j;
        intercambio(a, min, N);
        intercambio(info,min, N);
        p[info[min]] = min;
        return info[N--];
    }
    int vacio()
    { return (N <= 0); }
};
```

Esta implementación supone que las claves menores tienen las prioridades más altas, lo cual es por lo menos tan común en las aplicaciones como el esquema utilizado para la *ordenación por montículos*. La clave para valorar esta implementación es la operación cambiar. Como es habitual, no se hace comprobación de errores y se supone (por ejemplo) que *x* siempre está en el margen correcto y que el usuario no trata de insertar en una cola llena o suprimir en una cola vacía. La adición de código en C++ para tales controles se obtiene de forma directa.

Si la cola de prioridad es muy grande o se quiere hacer un gran número de operaciones de suprimir, o ambas cosas, el array *a* se debe mantener en el orden del montículo para asegurar que el tiempo de ejecución de todas las operaciones esté acotado por un factor logarítmico. Las rutinas del montículo dadas anteriormente se pueden modificar de forma directa para mantener los arrays *info* y *p* como se necesite: las comparaciones se refieren directamente a *a*, pero los desplazamientos se deben reflejar directamente en *info* e indirectamente en *p*, como en la implementación anterior. Todo esto se puede hacer sin cambiar la interfaz, por lo que más adelante se hará referencia a las operaciones sobre colas de prioridad como se definieron en esta clase, suponiendo que se pueden implementar eficazmente utilizando montículos tal y como se acaba de describir.

En algunas aplicaciones podrían ser apropiadas otras implementaciones, dependiendo de la naturaleza de las claves y de la mezcla de operaciones sobre colas de prioridad a llevar a cabo. Por ejemplo, si la cola de prioridad es más bien pequeña (por ejemplo, 20 elementos), la implementación anterior puede servir bastante bien. También pueden ser apropiados ligeros cambios en la interfaz. Por ejemplo, se puede desear que una función devuelva el valor de la clave de prioridad más alta de la cola y no precisamente el índice de la información asociada. O podría ser conveniente poder suprimir la clave más grande o la más pequeña de la cola. Es fácil añadir tales procedimientos a la clase anterior, pero es un desafío el desarrollar una clase donde se garantice un funcionamiento logarítmico en todas las operaciones.

Implementaciones avanzadas

Si se debe hacer la operación *unir* eficazmente, las implementaciones que se han hecho son insuficientes y se necesitan técnicas más avanzadas. Aunque no se dispone aquí de espacio para entrar en los detalles de tales métodos, se pueden presentar algunas consideraciones válidas para su diseño.

Por «eficazmente» se entiende que una *unión* se debe hacer al mismo tiempo que las otras operaciones. Esto excluye inmediatamente la representación sin enlaces que se ha venido utilizando para los montículos, puesto que dos de ellos sólo se pueden unir desplazando todos los elementos de al menos uno de ellos a un array mayor. Es fácil transformar los algoritmos estudiados para utilizar

representaciones enlazadas; de hecho, algunas veces existen razones para hacer esto (por ejemplo, puede no ser conveniente tener un array contiguo muy grande). En una representación directa por lista enlazada se tendría que mantener cada nodo apuntando a su padre y a sus dos hijos.

Esto revela que la propia condición del montículo parece ser demasiado fuerte para permitir implementaciones eficaces de la operación *unir*. Las estructuras de datos avanzadas que se han diseñado para resolver este problema debilitan o bien la condición del montículo o la del balance, en busca de obtener la flexibilidad que se necesita para la *unión*. Estas estructuras permiten que todas las operaciones se puedan hacer en tiempo logarítmico.

Ejercicios

1. Dibujar el montículo que se obtiene cuando se llevan a cabo las siguientes operaciones en un montículo inicialmente vacío: `insertar(1)`, `insertar(5)`, `insertar(2)`, `insertar(6)`, `sustituir(4)`, `insertar(8)`, `suprimir`, `insertar(7)`, `insertar(3)`.
2. ¿Es un montículo un archivo ordenado en orden inverso?
3. Decir cuál es el montículo que se obtiene cuando, comenzando con un montículo vacío, se llama sucesivamente a `insertar` para las claves C U E S T I O N F A C I L.
4. ¿Qué posiciones podrían estar ocupadas por la tercera clave más grande de un montículo de tamaño 32? ¿Qué posiciones no podrían estar ocupadas por la tercera clave más pequeña de un montículo de tamaño 32?
5. ¿Por qué no se utiliza un centinela para evitar la comprobación $j < N$ en `bajarmonticulo`?
6. Mostrar cómo se obtienen las funciones normales de pilas y colas en los casos particulares de colas de prioridad.
7. ¿Cuál es el número mínimo de claves que se deben desplazar en un montículo durante una operación de «suprimir el mayor»? Dibujar un montículo de tamaño 15 para el que se alcanza el mínimo.
8. Escribir un programa para eliminar el elemento de la posición d de un montículo.
9. Comparar empíricamente la construcción ascendente (de abajo hacia arriba) de un montículo con la descendente (de arriba hacia abajo), construyendo montículos con 1.000 claves aleatorias.
10. Dar el contenido de los arrays `p` e `info` después de insertar las claves C U E S T I O N F A C I L (siendo i la i -ésima letra de la serie) en un montículo inicialmente vacío.

Ordenación por fusión

En el Capítulo 9 se estudió la operación de *selección*, que permite encontrar el k -ésimo elemento más pequeño de un archivo, viéndose que es semejante a dividir el archivo en dos partes, los k elementos más pequeños y los $N-k$ más grandes. En este capítulo se examinará un proceso más o menos complementario, la *fusión*, que permite combinar dos archivos ordenados en otro más grande, también ordenado. Como se verá, la fusión es la base de un algoritmo de ordenación recursivo directo.

La selección y la fusión son operaciones complementarias en el sentido de que la primera divide el archivo en dos archivos independientes y la fusión une dos archivos independientes para hacer uno. La relación entre estas dos operaciones se hace evidente si se trata de aplicar el paradigma de «divide y vencerás» para crear un método de ordenación. El archivo puede estar distribuido de modo que cuando las dos partes estén ordenadas el archivo completo esté ordenado, o bien puede separarse en dos partes para ordenarlas y luego combinarlas para dejar ordenado el archivo completo. Ya se ha visto lo que sucede en el primer caso: es decir en el Quicksort, que consiste básicamente en un procedimiento de selección seguido de dos llamadas recursivas. A continuación se verá la ordenación por fusión, el complemento del Quicksort, que básicamente consiste en dos llamadas recursivas seguidas de un procedimiento de fusión.

La ordenación por fusión, al igual que la ordenación por montículos, tiene la ventaja de que ordena un archivo de N elementos en un tiempo proporcional a $N \log N$ aun en el peor caso. Su principal inconveniente es que parece difícil evitar la utilización de un espacio extra proporcional a N , a menos que se dedique un gran esfuerzo para superar este obstáculo. La longitud del bucle interno está entre la del Quicksort y la ordenación por montículos, por lo que la ordenación por fusión es una buena elección si la velocidad es lo esencial y hay espacio disponible. Más aún, la ordenación por fusión se puede implementar de forma que se pueda acceder secuencialmente a los datos (un elemento después de otro), lo que a veces es una cierta ventaja. Por ejemplo, la ordenación por fusión es el método ideal para ordenar una lista enlazada, en la que el acceso

secuencial es la única forma posible de acceso. Igualmente, como se verá en el Capítulo 13, la fusión es la base de la ordenación en dispositivos de acceso secuencial, aunque los métodos utilizados en ese contexto son algo diferentes de los empleados por la ordenación por fusión.

En muchos entornos de procesamiento de datos se mantiene un gran archivo (ordenado) de datos al que regularmente se le añaden nuevas entradas. Por lo regular, estas entradas nuevas se van «colocando en lotes» y concatenando al archivo principal (que es mucho más grande), reordenando luego el archivo completo. Esta situación está hecha a la medida de la fusión: una estrategia mucho mejor consiste en ordenar los lotes pequeños con las entradas nuevas y luego fusionarlos con el gran archivo principal. La fusión tiene muchas otras aplicaciones similares que hacen que su estudio merezca la pena. Se examinará también un método de ordenación basado en la fusión.

En este capítulo se concentrará el interés en los programas para *fusiones de dos vías*: programas que combinan dos archivos de entrada ordenados para producir un archivo ordenado de salida. En el próximo capítulo se verá con más detalle la *fusión multivía*, que implica más de dos archivos. (La aplicación más importante de la fusión multivía es la ordenación externa, el tema del presente capítulo.)

Para comenzar, se supone que se tienen dos arrays ordenados $a[1], \dots, a[M]$ y $b[1], \dots, b[N]$ de enteros que se quieren fusionar en un tercer array $c[1], \dots, c[M+N]$. El código siguiente es una implementación de la estrategia obvia que consiste en ir tomando sucesivamente para c el elemento más pequeño de los que van quedando en los arrays a y b :

```
i = 1; j = 1;
a[M+1] = elementoMAX; b[N+1] = elementoMAX;
for (k = 1; k <= M+N; k++)
    c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
```

La implementación se simplifica reservando espacio en los arrays a y b para las claves centinelas con valores mayores que cualquiera de las otras claves. Cuando se termine con el array a (b) el bucle simplemente desplaza el resto de los elementos del array b (a) al array c . Este método utiliza obviamente $M + N$ comparaciones. Si $a[M+1]$ y $b[N+1]$ no pudieran utilizarse por las claves centinelas, entonces habría que añadir comprobaciones para estar seguros de que i es siempre menor que M y que j es menor que N . Otra forma de evitar esta dificultad es la que se utiliza posteriormente en la implementación de la ordenación por fusión.

En lugar de utilizar un espacio extra proporcional al tamaño del archivo fusionado, sería preferible tener un método *in situ* que utilice $c[1], \dots, c[M]$ para una entrada y $c[M+1], \dots, c[M+N]$ para la otra. A primera vista parece fácil de hacer, pero no es así: tales métodos existen pero son tan complicados que incluso una *ordenación in situ* probablemente sea más eficaz, a menos que se les dedique un gran cuidado. Se volverá sobre este punto más adelante.

Puesto que en implementaciones prácticas se necesita espacio extra, se podrían considerar implementaciones con listas enlazadas. De hecho, este método es ideal para estas estructuras. A continuación se da una implementación completa que ilustra todos los convenios a utilizar; obsérvese que el código para la fusión es casi tan sencillo como el anterior:

```
struct nodo
    { TipoElemento clave; struct nodo *siguiente; };
struct nodo *z;
struct nodo *fusion(struct nodo *a, struct nodo *b)
{
    struct nodo *c;
    c = z;
    do
        if (a->clave <= b->clave)
            { c->siguiente = a; c = a; a = a->siguiente; }
        else
            { c->siguiente = b; c = b; b = b->siguiente; }
    while (c != z);
    c = z->siguiente; z->siguiente = z;
    return c;
}
```

Este programa fusiona las listas a las que apuntan a y b con la ayuda de un puntero auxiliar c.

En este capítulo se tratarán directamente enlaces sobre las listas en lugar de utilizar la clase Lista del Capítulo 3 para economizar a la hora de expresar los algoritmos de ordenación por fusión, como resultará evidente más abajo. Se supone que las listas tienen un nodo ficticio «cola», como en el Capítulo 3: todas las listas terminan con el nodo ficticio z, el cual normalmente apunta a sí mismo y también sirve como centinela, con $z->\text{clave} == \text{elemento MAX}$. Durante la fusión, z se utiliza para contener el primer elemento de la nueva lista fusionada (esto es, utilizándolo como nodo *cabecera* cuyo campo siguiente apunta al principio de la lista). Después de construir la lista fusionada, el puntero a su primer nodo está dado por $z->\text{siguiente}$ y z se reinicializa para que se apunte a sí mismo.

La comparación de clave de fusión incluye la igualdad, de forma que la

fusión será estable si se considera que la lista b sigue a la lista a. Más adelante se verá cómo esta estabilidad puede transmitirse a los programas de ordenación que utilizan la fusión.

Ordenación por fusión

Una vez que se tiene un procedimiento de fusión, no es difícil utilizarlo como base de un procedimiento recursivo de ordenación. Para ordenar un archivo dado, se divide en dos, se ordenan las dos mitades (recursivamente) y se fusionan entre sí. La implementación siguiente de este proceso ordena un array $a[izq], \dots, a[der]$ (utilizando un array auxiliar $b[izq], \dots, b[der]$):

```
void ordenfusion( TipoElemento a[], int izq, int der)
{
    int i, j, k, m;
    if (der > izq)
    {
        m = (der+izq)/2;
        ordenfusion(a, izq, m);
        ordenfusion(a, m+1, der);
        for (i = m+1; i > izq; i--) b[i-1] = a[i-1];
        for (j = m; j < der; j++) b[der+m-j] = a[j+1];
        for (k = izq; k <= der; k++)
            a[k] = (b[i]<b[j]) ? b[i++] : b[j--];
    }
}
```

Este programa efectúa la fusión sin utilizar centinelas copiando el segundo array de forma simétrica al primero, pero en orden inverso. Así cada array sirve como «centinela» para el otro: el elemento más grande (que se encuentra en un array o en el otro) garantiza desarrollos adecuados una vez agotado el otro array al hacer la fusión. El «bucle interno» de este programa es bastante corto (mover hacia b, mover de nuevo hacia a, incrementar i o j e incrementar y comprobar k), y podría acortarse aún más teniendo dos copias del código (una para fusionar a en b y otra para fusionar b en a), aunque esto requeriría volver a utilizar de nuevo los centinelas.

El archivo ejemplo de claves se procesa como se muestra en la Figura 12.1. Cada línea muestra el resultado de una llamada a `fusion`. Primero se fusionan E y J para obtener E J, luego E y M para obtener E M y éstas con E J para obtener E E J M. Luego se fusiona L P con A O para obtener A L O P, que fusionado con E E J M da A E E J L M O P, etc. Así pues, este método cons-

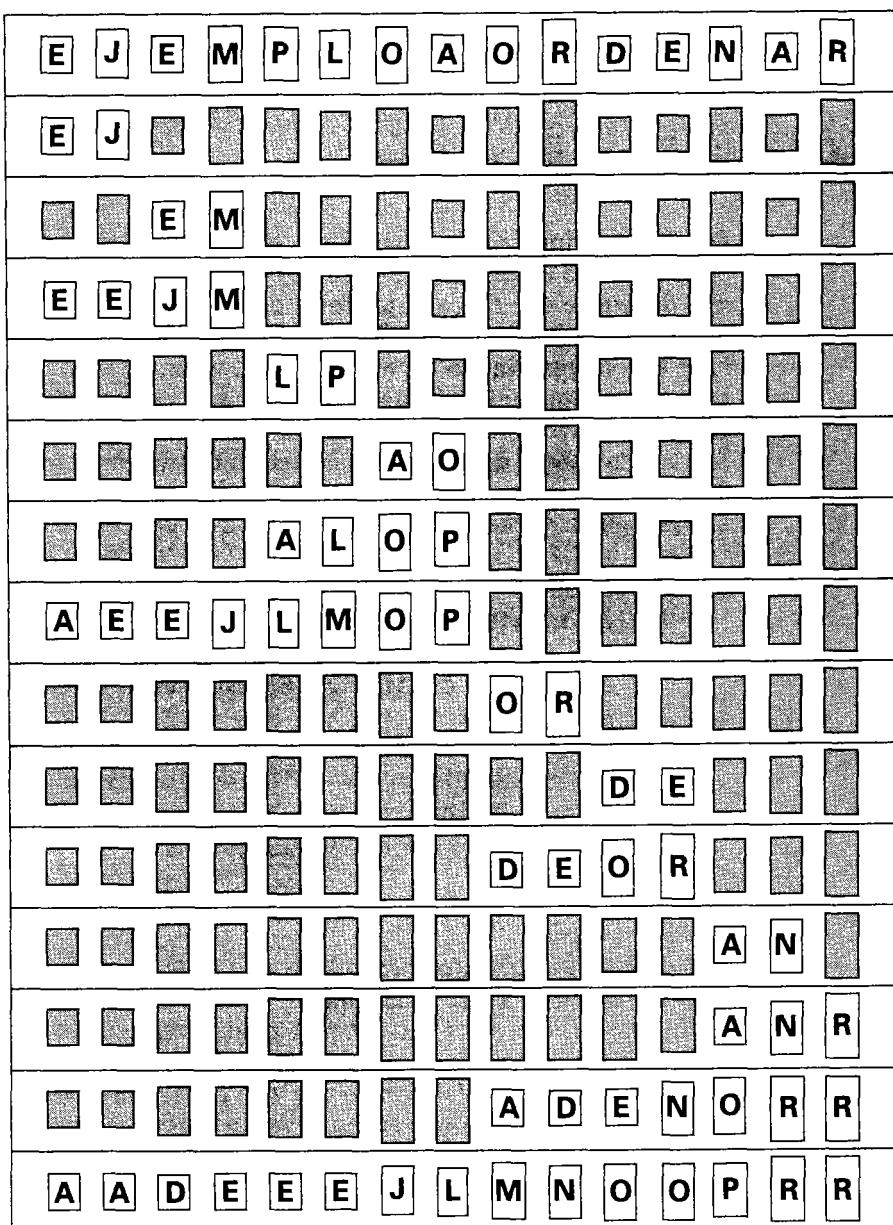


Figura 12.1 Ordenación por fusión recursiva.

truye recursivamente archivos ordenados a partir de archivos ordenados más pequeños.

Ordenación por fusión de listas

Este proceso implica un movimiento tal de datos que también se debe considerar una lista enlazada. El programa que sigue es una implementación recursiva directa de una función que toma como entrada un puntero a una lista no ordenada y devuelve un puntero a la versión ordenada de la lista. El programa hace esto reorganizando los nodos de la lista sin necesidad de asignar espacio para nodos temporales ni listas. (Es conveniente pasar como parámetro al programa recursivo la longitud de la lista; también se puede almacenar este valor con la lista o dejar que el programa recorra la lista para averiguar tal longitud.)

```
struct nodo *ordenfusion(struct nodo *c)
{
    struct nodo *a, *b;
    if (c->siguiente != z)
    {
        a = c; b = c->siguiente->siguiente->siguiente;
        while (b != z)
            { c = c->siguiente; b = b->siguiente->siguiente; }
        b = c->siguiente; c->siguiente = z;
        return fusion(ordenfusion(a), ordenfusion(b));
    }
    return c;
}
```

Este programa ordena dividiendo la lista sobre la que apunta c en dos mitades, a las que apuntan a y b, ordenando después las dos mitades recursivamente y utilizando a continuación `fusion` para producir el resultado final. Una vez más, este programa se adhiere al convenio de considerar que todas las listas terminan con `z`: la lista de entrada termina con `z` (y por lo tanto esto hace que la lista b también), y la instrucción explícita `c->siguiente = z` pone `z` al final de la lista a. Este programa es bastante fácil de comprender en su formulación recursiva, aun cuando realmente es un algoritmo sofisticado.

Ordenación por fusión ascendente

Como se presentó en el Capítulo 5, todo programa recursivo tiene un análogo no recursivo, que, aunque equivalente, puede ejecutar las operaciones en un or-

den diferente. En realidad, la ordenación por fusión es un prototipo de la estrategia de «combina y vencerás» que caracteriza a muchos cálculos de este tipo, por lo que merece la pena estudiar detalladamente sus implementaciones no recursivas.

La versión más simple de la ordenación por fusión no recursiva procesa un conjunto de archivos ligeramente diferentes en un orden diferente: primero recorre la lista llevando a cabo una fusión 1 por 1 para producir sublistas de tamaño 2, luego recorre la lista llevando a cabo fusiones 2 por 2 para producir sublistas ordenadas de tamaño 4, luego hace fusiones 4 por 4 para producir sublistas de tamaño 8, etc., hasta que se ordene la lista completa.

La Figura 12.2 muestra cómo este método lleva a cabo esencialmente las mismas fusiones que en la Figura 12.1 para el archivo ejemplo (puesto que su tamaño está próximo a una potencia de dos), pero en un orden diferente. En general, se necesitan $\log N$ pasadas para ordenar un archivo de N elementos, pues en cada pasada se duplica el tamaño de los subarchivos ordenados.

Es importante notar que las fusiones reales efectuadas por este método «ascendente» no son las mismas que las realizadas por la implementación anterior. Considérese la ordenación de 95 elementos que se muestra en la Figura 12.3. La última fusión es una 64 por 31, mientras que en la ordenación recursiva sería un 47 por 47. Es posible, sin embargo, organizar las cosas de forma que la secuencia de fusiones hecha por los dos métodos sea la misma, aunque no hay ninguna razón particular para hacer esto.

A continuación se da una implementación detallada de esta aproximación ascendente, utilizando listas enlazadas.

```
struct nodo *ordenfusion(struct nodo *c)
{
    int i, N;
    struct nodo *a, *b, *cabeza, *resto, *t;
    cabeza = new nodo;
    cabeza->siguiente = c; a = z;
    for (N = 1; a != cabeza->siguiente; N = N+N)
    {
        resto = cabeza->siguiente; c = cabeza;
        while (resto != z)
        {
            t = resto; a = t;
            for (i = 1; i < N; i++) t = t->siguiente;
            b = t->siguiente; t->siguiente = z; t = b;
            for (i = 1; i < N; i++) t = t->siguiente;
            resto = t->siguiente; t->siguiente = z;
            c->siguiente = fusion(a, b);
            for (i = 1; i <= N+N; i++) c = c->siguiente;
        }
    }
}
```

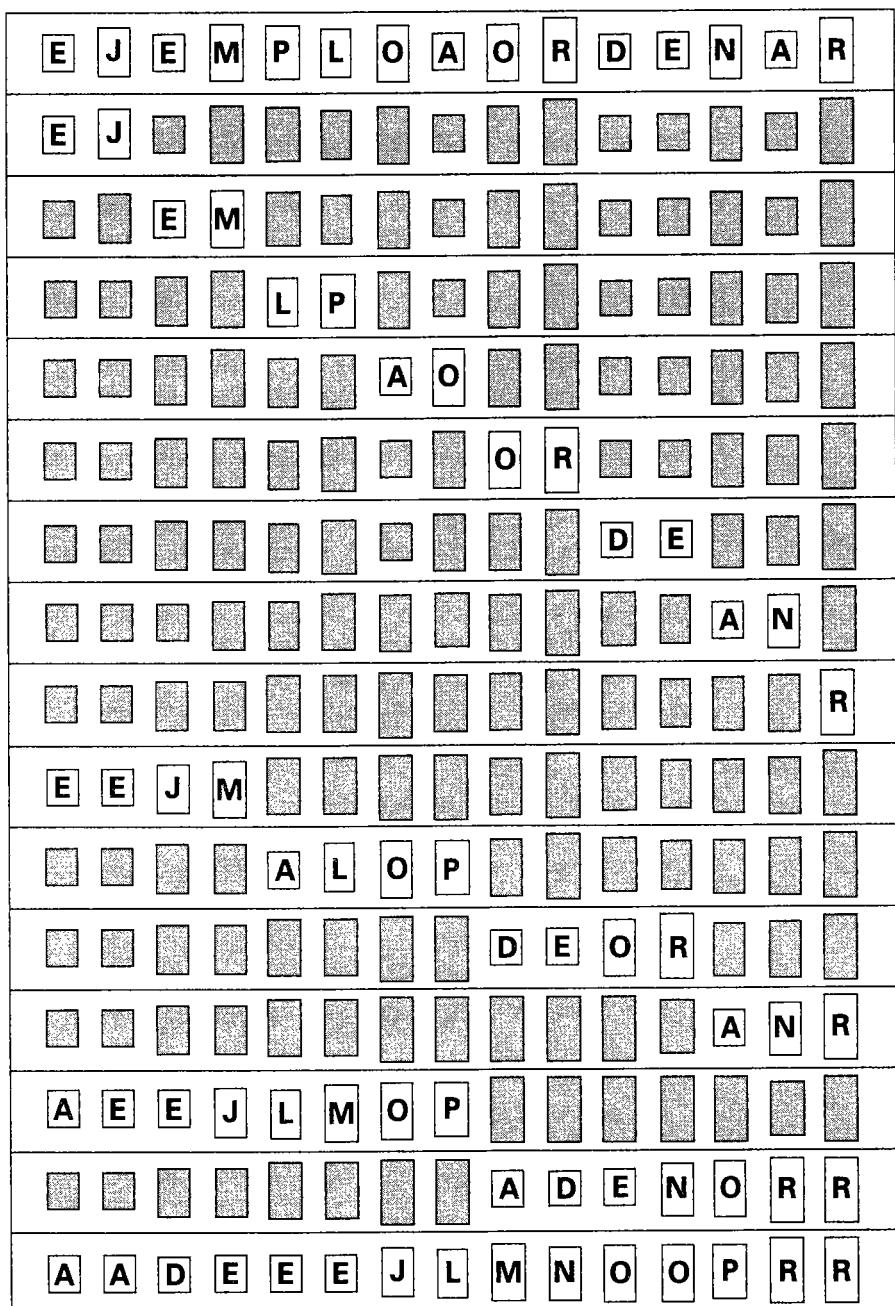


Figura 12.2 Ordenación por fusión no recursiva.

```
    }  
}  
return cabeza->siguiente;  
}
```

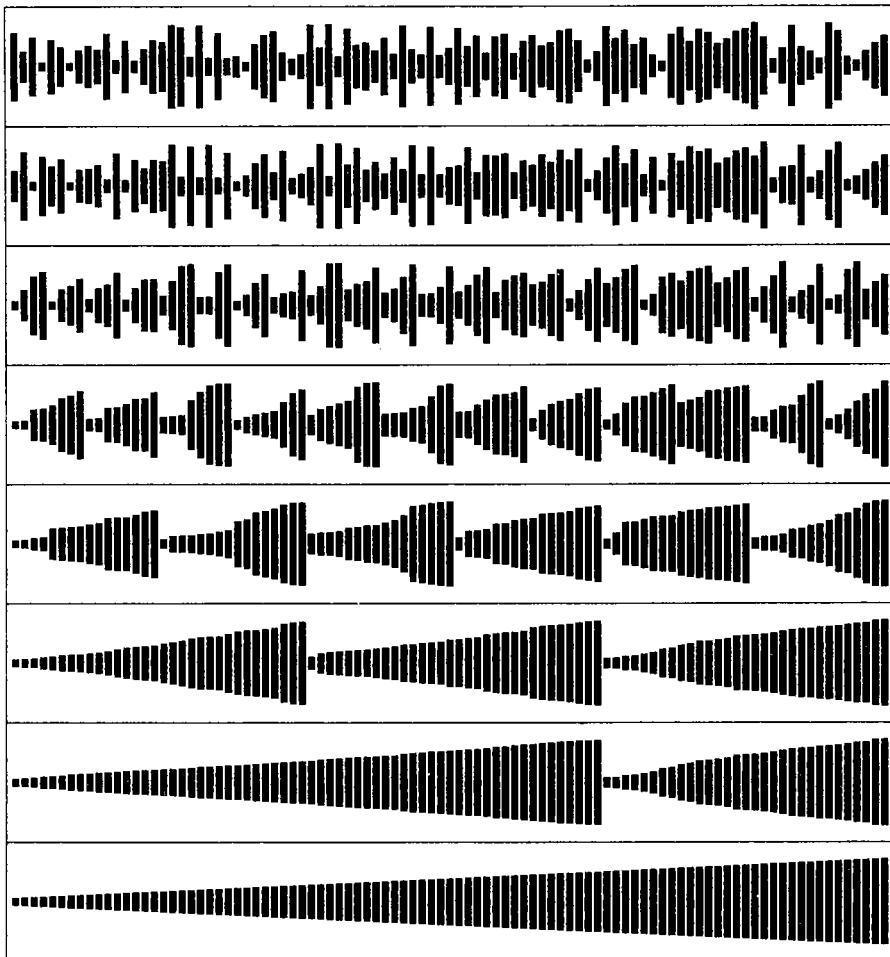


Figura 12.3 Ordenación por fusión de una permutación aleatoria.

Este programa utiliza un nodo «cabecera de lista» (al que apunta `cabeza`) cuyo campo enlace apunta a la lista enlazada que se está ordenando. Cada iteración del bucle externo (`for`) recorre el archivo, produciendo una lista enlazada compuesta de subarchivos ordenados, dos veces más grandes que los de la pasada anterior. Esto se hace manteniendo dos punteros, uno a la parte de la lista que

aún no se ha visto (*resto*) y otro al final de la parte de la lista en la que ya se han fusionado los subarchivos (*c*). El bucle interno (*while*) fusiona los dos subarchivos de longitud N comenzando con el nodo al que apunta *resto* y produce un subarchivo de longitud $N+N$, el cual se enlaza con la lista *c* del resultado.

La fusión real se lleva a cabo almacenando un enlace al primer subarchivo a fusionar en *a*, saltando luego N nodos (utilizando el enlace temporal *t*), y enlazando *z* con el final de la lista de *a*, haciendo después lo mismo para tener otra lista de N nodos a la que apunta *b* (actualizando *resto* con el enlace al último nodo visitado) y llamando después *fusion*. (Entonces se actualiza *c* siguiendo hacia abajo hasta el final de la lista que se acaba de fusionar. Éste es un método más simple, pero algo menos eficaz, que las diversas alternativas disponibles, tales como hacer que *fusion* devuelva punteros al principio y al final o mantener múltiples punteros sobre cada nodo de la lista.)

La ordenación por fusión ascendente es también un método interesante para utilizar en una implementación con arrays; esto se deja al lector como un ejercicio instructivo.

Características de rendimiento

La ordenación por fusión es importante porque es un método de ordenación «óptimo» bastante directo que se puede implementar de forma estable. Estos hechos son relativamente fáciles de demostrar.

Propiedad 12.1 *La ordenación por fusión necesita alrededor de $N\lg N$ comparaciones para ordenar un archivo de N elementos.*

En la implementación anterior, cada fusión M por N necesitará $M + N$ comparaciones (esto podría variar en una o dos unidades, dependiendo de cómo se utilizan los centinelas). En una ordenación por fusión ascendente, se utilizan $\lg N$ pasadas y cada una necesita alrededor de N comparaciones. Para la versión recursiva, el número de comparaciones se describe por la recurrencia estándar de «divide y vencerás» $M_N = 2M_{N/2} + N$, con $M_1 = 0$. Se sabe del Capítulo 6 que esta ecuación admite la solución $M_N \approx N\lg N$. Precisamente estos argumentos son los dos verdaderos si N es una potencia de dos; se deja como ejercicio demostrar que esto ocurre también para cualquier N . Más aún, esto también es válido en el caso medio.■

Propiedad 12.2 *La ordenación por fusión utiliza un espacio extra proporcional a N .*

Esto se deduce de las implementaciones, pero se pueden dar algunos pasos para disminuir el impacto de este problema. Por supuesto, si el «archivo» a ordenar

es una lista enlazada, el problema no aparece, dado que el «espacio extra» (para los enlaces) está por otros motivos.

Para arrays, es fácil hacer una fusión M por N utilizando espacio extra solamente para el más pequeño de los dos arrays (ver Ejercicio 2). Esto reduce a la mitad las necesidades de espacio de la ordenación por fusión. En realidad es posible hacer esto mucho mejor y hacer fusiones *in situ*, aunque en la práctica es poco probable que merezca la pena.■

Propiedad 12.3 *La ordenación por fusión es estable.*

Puesto que todas las implementaciones realmente sólo mueven las claves durante las fusiones, es suficiente verificar que las fusiones en sí son estables. Pero esto es evidente: la posición relativa de las claves iguales no se altera por el proceso de fusión.■

Propiedad 12.4 *La ordenación por fusión es insensible al orden inicial de la entrada.*

En las implementaciones, la entrada determina sólo el orden en el que se procesan los elementos en las fusiones, por lo tanto esta sentencia es literalmente exacta (excepto para alguna variación que depende de cómo se compila y ejecuta la instrucción `if`, lo que debería ser insignificante). Otras implementaciones de fusión, que implican comprobaciones relativas al primer archivo que se recorra completamente, pueden conducir a algunas variaciones más grandes, según sea la entrada, pero no mucho mayores. El número de pasadas que se necesita depende sólo del tamaño del archivo, no de su contenido, y cada pasada necesita alrededor de N comparaciones (realmente $N \cdot O(1)$ como media, como se explicará más adelante). Pero el peor caso es más o menos el mismo que el caso medio.■

La Figura 12.4 muestra una ordenación por fusión ascendente que opera sobre un archivo que está inicialmente en orden inverso. Es interesante comparar esta figura con la Figura 8.9, que muestra a la ordenación de Shell haciendo las mismas operaciones.

La Figura 12.5 presenta otro aspecto de la ordenación por fusión que opera sobre una permutación aleatoria, para compararla con diagramas similares de los primeros capítulos. En particular la Figura 12.5 muestra una sorprendente semejanza con la Figura 10.5: en este sentido, la ordenación por fusión es ¡la «transpuesta» del método de ordenación por residuos!

Implementaciones optimizadas

En la presentación de los centinelas ya se ha prestado alguna atención al bucle interno de la ordenación por fusión basado en arrays, y se ha visto que las com-

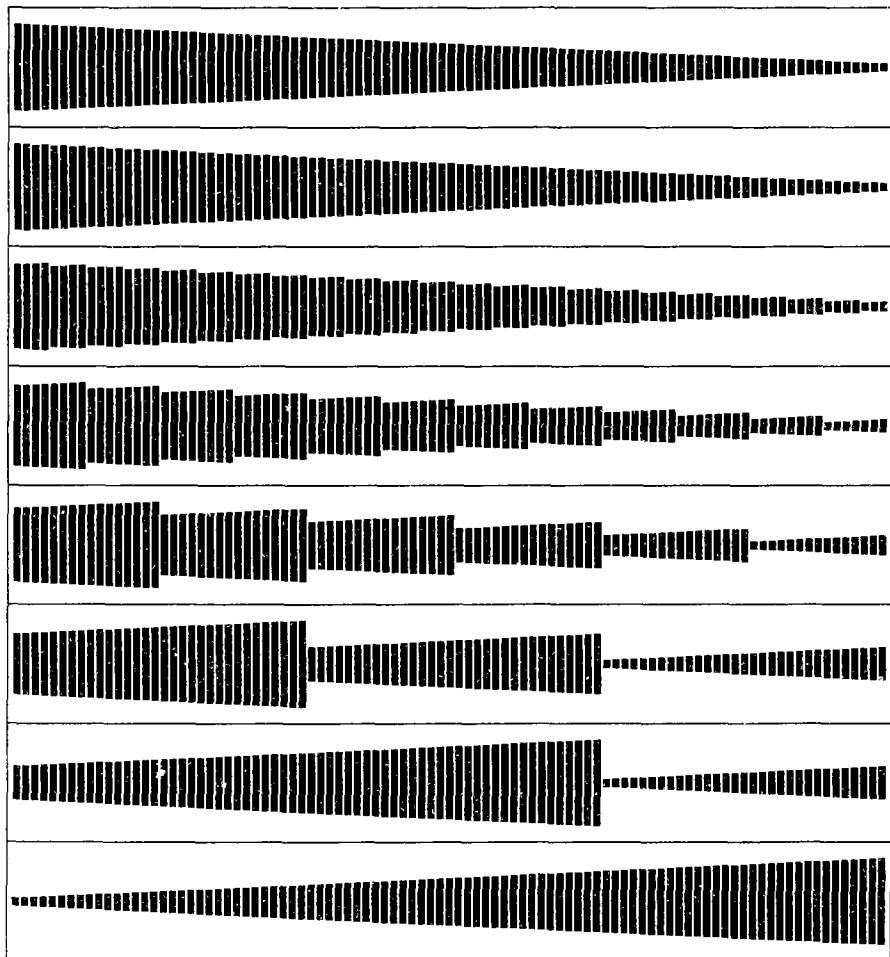


Figura 12.4 Ordenación por fusión de una permutación en orden inverso.

probaciones de los límites del array en el bucle interno se pueden evitar invirtiendo el orden de uno de los arrays. Esto llama la atención sobre una de las mayores deficiencias de la implementación anterior: el desplazamiento de a hacia b . Como se vio para el método de ordenación por residuos del Capítulo 10, este desplazamiento se puede evitar utilizando dos copias del código, una para fusionar de a a b y otra de b a a .

Para llevar a cabo una combinación de estas dos mejoras, es necesario cambiar las cosas de modo que *fusion* pueda dar como salida los arrays, en orden creciente o decreciente. En la versión no recursiva, esto se lleva a cabo alternando entre una salida creciente y una decreciente; en la versión recursiva hay

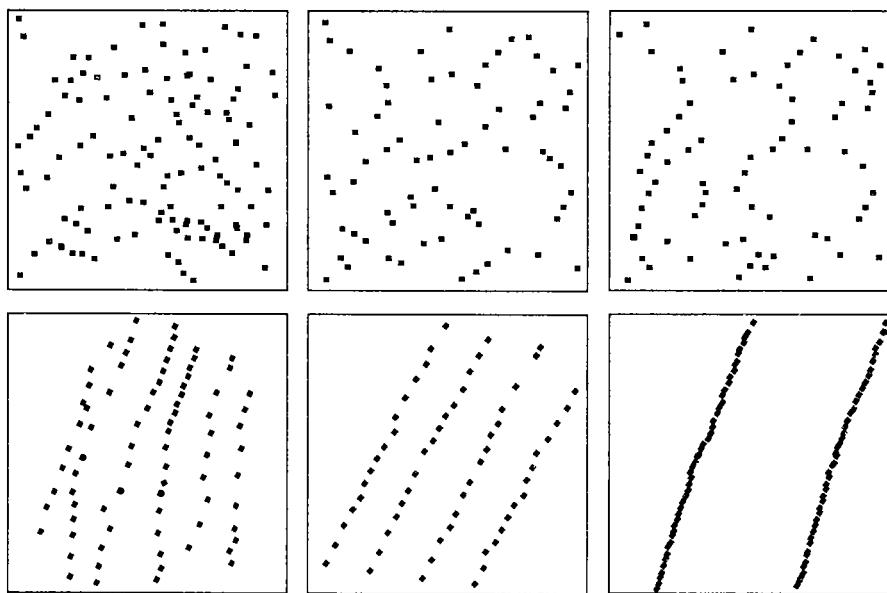


Figura 12.5 Ordenación por fusión de una permutación aleatoria.

que tener cuatro rutinas recursivas para fusionar $a(b)$ en $b(a)$ con el resultado en orden decreciente o creciente. Cualquiera de ellos reducirá el bucle interno de la ordenación por fusión a una comparación, un almacenamiento, dos incrementos de punteros (i o j , y k) y una comprobación del puntero. Esto compite favorablemente con una comparación, un incremento y una comprobación y un intercambio (parcial) del Quicksort, y el bucle interno del Quicksort se ejecuta $2\ln N \approx 1,38\lg N$ veces, alrededor de un 38% más frecuentemente que el de la ordenación por fusión.

Revisión de la recursión

Los programas de este capítulo, junto con el del Quicksort, son implementaciones típicas de los algoritmos de divide y vencerás. En capítulos posteriores se verán varios algoritmos con estructuras similares, por lo que merece la pena echar un vistazo más detallado a algunas de las características básicas de estas implementaciones.

El Quicksort es realmente un algoritmo de «vence y dividirás»: en una implementación recursiva, la mayor parte del trabajo se hace *antes* de las llamadas recursivas. Por el contrario, la ordenación por fusión recursiva está más en el espíritu de «divide y vencerás»: cada archivo se divide primero en dos partes y

luego se «vence» a cada una individualmente. El primer problema para el que la ordenación por fusión hace el procesamiento real es el más pequeño; el subarchivo mayor se procesa al final. En el Quicksort el procesamiento comienza sobre el subarchivo mayor y finaliza con los más pequeños.

Esta diferencia se manifiesta en las implementaciones no recursivas de los dos métodos. El Quicksort debe mantener una pila, puesto que tiene que memorizar grandes subproblemas, que se dividen en función de los datos. La ordenación por fusión admite una versión no recursiva simple porque la forma en que divide el archivo es independiente de los datos, por lo que el orden en el que procesa los subproblemas puede reorganizarse de alguna forma para hacer el programa más simple.

Otra diferencia práctica que se manifiesta por sí misma es que la ordenación por fusión es estable (está implementada adecuadamente) y el Quicksort no lo es (sin tener que recurrir a complicaciones extra). Para la ordenación por fusión, si se supone (por inducción) que los subarchivos han sido ordenados establemente, es suficiente con asegurar que la fusión se hace de una manera estable, lo que puede hacerse con facilidad. Pero para el Quicksort no parece existir, por sí misma, ninguna forma fácil de hacer la partición de una manera estable, lo que impide la posibilidad de estabilidad, incluso antes de que entre en juego la recursión.

Una nota final: como el Quicksort o cualquier otro programa recursivo, la ordenación por fusión se puede mejorar tratando los subarchivos pequeños de forma diferente. En las versiones recursivas del programa esto se puede implementar exactamente como para el Quicksort, bien haciendo sobre la marcha una ordenación por inserción de los subarchivos pequeños, bien haciendo una pasada final de limpieza. En las versiones no recursivas, los pequeños subarchivos ordenados se pueden construir en una pasada inicial utilizando una versión modificada de la ordenación por inserción o por selección. Otra idea que se ha sugerido para la ordenación por fusión es aprovecharse del orden «natural» del archivo utilizando un método ascendente para fusionar las dos primeras secuencias ordenadas del archivo (sin importar lo largas que puedan ser), después las dos secuencias siguientes, etc., repitiendo el proceso hasta que el archivo quede ordenado. A pesar de lo atractiva que pueda parecer esta idea, no se puede comparar con el método estándar que se ha presentado, porque el coste de identificar las secuencias, que debe imputarse al bucle interno, es mayor que las ganancias alcanzadas, excepto para ciertos casos degenerados (tales como un archivo ya ordenado).

Ejercicios

1. Implementar una ordenación por fusión recursiva, que procese por medio de una ordenación por inserción los subarchivos con menos de M elemen-

- tos; determinar empíricamente el valor de M para el que se ejecuta más rápidamente sobre un archivo aleatorio de 1.000 elementos.
2. Comparar empíricamente la ordenación por fusión recursiva y la no recursiva para listas enlazadas y $N = 1.000$.
 3. Implementar la ordenación por fusión recursiva para un array de N enteros, utilizando un array auxiliar de tamaño menor que $N/2$.
 4. Verdadero o falso: el tiempo de ejecución de la ordenación por fusión no depende del valor de las claves del archivo de entrada. Explicar la respuesta.
 5. ¿Cuál es el número mínimo de pasos de la ordenación por fusión (dentro de un factor constante)?
 6. Implementar una ordenación por fusión ascendente no recursiva que utilice dos arrays en lugar de listas enlazadas.
 7. Mostrar las fusiones efectuadas al utilizar la ordenación por fusión recursiva para ordenar las claves C U E S T I O N F A C I L.
 8. Mostrar el contenido de las listas enlazadas de cada iteración al utilizar la ordenación por fusión no recursiva para ordenar las claves C U E S T I O N F A C I L.
 9. Intentar escribir una ordenación por fusión recursiva, utilizando arrays, partiendo de la idea de hacer fusiones de tres vías en lugar de dos vías.
 10. Comprobar empíricamente, para archivos aleatorios de tamaño 1.000, la afirmación hecha en el texto de que no compensa la idea de aprovecharse del orden «natural» en el archivo.

Ordenación externa

Muchas importantes aplicaciones de ordenación deben procesar archivos muy grandes, demasiado como para tenerlos en la memoria principal de cualquier computadora. Los métodos adaptados a estas aplicaciones se denominan métodos *externos*, puesto que implican un gran volumen de procesamiento externo a la unidad central de proceso (en contraste con los métodos *internos* que se han visto anteriormente).

Hay dos factores determinantes que hacen que los algoritmos externos sean diferentes de los que se han visto hasta ahora. El primero es que el coste de acceso a un elemento es infinitamente más grande que el de cualquier actualización o cálculo. El segundo, y todavía más costoso que el anterior, es que existen severas restricciones de acceso, dependiendo del medio de almacenamiento externo utilizado: por ejemplo, no se puede acceder a los elementos de una cinta magnética más que de forma secuencial.

La gran variedad de dispositivos de almacenamiento externo y de costes hacen que el desarrollo de los métodos de ordenación externos sea muy dependiente de la tecnología actual. Estos métodos pueden ser muy complicados y son numerosos los parámetros que afectan su rendimiento: un método muy ingenioso puede que no sea apreciado o utilizado por un simple cambio en la tecnología. Por esta razón este capítulo se centra más en los métodos generales que en el desarrollo de implementaciones específicas.

En síntesis, tratándose de la ordenación externa, los aspectos del problema relativos al «sistema» son tan importantes como los aspectos «algorítmicos». Ambas áreas se deben considerar cuidadosamente si se quiere desarrollar una ordenación externa eficaz. El coste principal de la ordenación externa se debe a la entrada-salida. Un buen ejercicio para alguien que planea hacer un programa para ordenar un archivo muy grande es implementar antes un programa para copiar un gran archivo y luego (si esto ha sido demasiado fácil) implementar un programa eficaz para invertir el orden de los elementos de un archivo de este tipo. Los problemas de sistema que aparecen al tratar de resolver estos problemas eficazmente son similares a los que aparecen en las ordenaciones externas.

Permutar un gran archivo externo de forma no trivial es tan difícil como ordenarlo, aun cuando no se necesiten comparaciones entre claves, etc. En la ordenación externa, se desea principalmente limitar el número de veces que cada elemento de datos se desplaza entre el medio de almacenamiento externo y la memoria principal, y estar seguro de que tales transferencias se hacen tan eficazmente como lo permita el material del que se dispone.

Se han desarrollado métodos de ordenación externa que se adaptan a las tarjetas perforadas y cintas de papel del pasado, a las cintas magnéticas y discos del presente y a las nuevas tecnologías como las memorias de burbuja y los videodiscos. La diferencia esencial entre los múltiples dispositivos son el tamaño del almacenamiento disponible y la velocidad y los tipos de restricción de acceso a los datos. En este libro se estudian los principios básicos de ordenación en las cintas magnéticas y los discos, porque estos dispositivos son posiblemente los que continuarán siendo muy utilizados e ilustran los dos modos fundamentales de acceso que caracterizan a muchos sistemas de almacenamiento externo. Frequentemente los sistemas modernos tienen una «jerarquía de almacenamiento» de varias memorias cada vez más lentas, baratas y voluminosas. Aunque muchos de los algoritmos que se van a considerar pueden transformarse en algoritmos eficaces en tales entornos, aquí se tratarán exclusivamente las memorias de «dos niveles» de jerarquía, que comprenden una memoria principal y una de disco o cinta.

Ordenación-fusión

La mayoría de los métodos de ordenación externa utilizan la siguiente estrategia general: primero hacen una pasada a lo largo del archivo a ordenar, dividiendo a éste en bloques del tamaño de la memoria interna, y *ordenando* estos bloques. Luego *fusionan* entre sí los bloques ordenados haciendo varias pasadas a través del archivo, creando sucesivamente archivos ordenados más grandes hasta que el archivo completo esté ordenado. El acceso a los datos es en su mayoría de forma secuencial, lo que hace apropiado este método para la mayor parte de los dispositivos externos. Los algoritmos de ordenación externa intentan reducir el número de pasadas sobre el archivo y aproximar lo máximo posible el coste de una pasada sencilla al coste de una copia.

Puesto que la mayor parte del coste de un método de ordenación externa se debe a la entrada-salida, es posible tener una medida aproximada del coste de una ordenación-fusión contando el número de veces que se lee o escribe una palabra del archivo (el número de pasadas sobre todos los datos). En muchas aplicaciones los métodos que se van a considerar implican unas diez pasadas o menos, lo que supone que cualquier método que pueda eliminar aunque sólo sea una simple pasada es digno de interés. Además, el tiempo de ejecución de la ordenación externa global puede estimarse fácilmente a partir del tiempo de

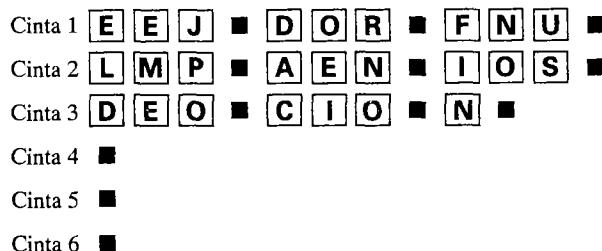


Figura 13.1 Fusión equilibrada de tres vías: resultado de la primera pasada.

ejecución de la acción de «invertir el archivo de copia», ejercicio sugerido anteriormente.

Fusión múltiple equilibrada

Para empezar, se seguirán los diferentes pasos del procedimiento más simple de ordenación-fusión sobre un ejemplo pequeño. Se supone que los registros con las claves E J E M P L O D E O R D E N A C I O N F U S I O N en una cinta de entrada se deben ordenar y colocar sobre una cinta de salida. Utilizar una «cinta» significa simplemente la obligación de leer los registros secuencialmente: el segundo registro no puede leerse hasta que no se haya leído el primero, y así sucesivamente. Se supone además que sólo hay espacio en la memoria para tres registros, pero que se dispone de todas las cintas que se deseé.

El primer paso consiste en leer del archivo tres registros cada vez, ordenarlos en bloques de tres registros y dar como salida los bloques ordenados. Así, primero se lee E J E y se obtiene el bloque E E J, luego se lee M P L, dando como salida al bloque L M P, y así sucesivamente. Ahora bien, para que estos bloques se puedan fusionar entre sí, deben estar en cintas diferentes. Si se desea hacer una fusión de tres vías, entonces se deben utilizar tres cintas, finalizando la ordenación anterior con la configuración que se muestra en la Figura 13.1.

Ahora ya se pueden fusionar los bloques ordenados de tamaño tres. Se lee el primer registro de cada cinta de entrada (hay espacio justo para ello en la memoria) y se extrae el de menor clave. A continuación se lee el siguiente registro de la misma cinta de la que se leyó el que se acaba de extraer y, de nuevo, se da salida al registro de la memoria que tenga la menor clave. Cuando se alcance el final de un bloque de tres palabras de una de las entradas, entonces se ignora esa cinta hasta que se hayan procesado los respectivos bloques de las otras dos y se haya dado salida a nueve registros. Luego se repite el proceso para fusionar los segundos bloques de tres palabras de cada cinta en un nuevo bloque de nueve palabras (que se escribe en una cinta diferente, para que esté listo para la pró-

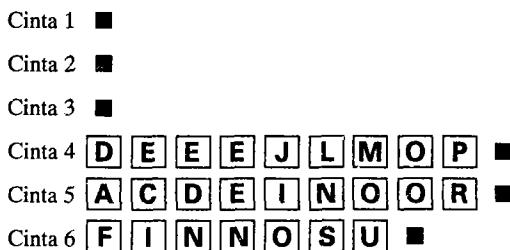


Figura 13.2 Fusión equilibrada de tres vías: resultado de la segunda pasada.

xima fusión). Continuando de esta forma, se llega a los tres grandes bloques configurados como muestra la Figura 13.2.

Ahora una última fusión de tres vías completa la ordenación. Si se tiene un archivo mucho mayor con múltiples bloques de tamaño 9 en cada cinta, entonces se finaliza la segunda pasada con bloques de tamaño 27 en las cintas 1, 2 y 3, y una tercera pasada produciría bloques de tamaño 81 en las cintas 4, 5 y 6, y así sucesivamente. Se necesitan seis cintas para ordenar un archivo arbitrariamente grande: tres para la entrada y tres para la salida de cada fusión de tres vías. (Realmente, se puede hacer con sólo cuatro cintas: la salida se puede colocar sólo en una cinta y después distribuir los bloques de ella sobre las tres cintas de entrada entre cada pasada de fusión.)

Este método se denomina *fusión múltiple equilibrada*: constituye un algoritmo razonable para hacer ordenaciones externas y es un buen punto de partida para una implementación de ordenación externa. Los algoritmos más sofisticados que se describen después pueden hacer que la ordenación se ejecute algo más rápidamente, pero no mucho más. (Sin embargo, cuando los tiempos de ejecución se miden en horas, lo que no es raro en la ordenación externa, incluso un pequeño tanto por ciento de disminución del tiempo de ejecución puede ser importante.)

Suponiendo que la ordenación debe emplear N palabras y que se dispone de una memoria interna de tamaño M , cada pasada de «ordenación» produce alrededor de N/M bloques ordenados. (Esta estimación supone registros de una palabra: para registros más grandes, el número de bloques ordenados se calcula multiplicando el resultado anterior por el tamaño del registro.) Si se hacen fusiones de P -vías en cada paso posterior, el número de pasadas es alrededor de $\log_p(N/M)$, puesto que cada paso reduce el número de bloques ordenados en un factor P .

Aunque los pequeños ejemplos pueden ayudar a comprender los detalles del algoritmo, es mejor razonar en términos de archivos muy grandes cuando se trabaja con ordenaciones externas. Por ejemplo, la fórmula anterior indica que si se utiliza una fusión de cuatro vías para ordenar un archivo de 200 millones de palabras en una computadora de un millón de palabras de memoria, el nú-

mero de pasadas podría ser aproximadamente cinco. Se puede obtener una estimación muy grosera del tiempo de ejecución multiplicando por cinco el correspondiente a una implementación de ordenación en orden inverso sugerida con anterioridad.

Selección por sustitución

La implementación del método anterior se puede desarrollar de una forma muy elegante y eficaz utilizando colas de prioridad. En primer lugar se verá que las colas de prioridad ofrecen una forma natural de implementar una fusión múltiple. Más importante aún es que se pueden utilizar las colas de prioridad para la pasada inicial de ordenación de forma tal que produzcan bloques ordenados mucho más grandes que los que puede contener la memoria interna.

La operación básica que se necesita para hacer una fusión de P -vías es dar salida al más pequeño de los elementos más pequeños todavía presentes en cada uno de los P bloques a fusionar. Ese elemento más pequeño debería reemplazarse por el siguiente elemento del bloque del que proviene. La operación *sustituir* en una cola de prioridad de tamaño P es exactamente lo que se necesita. (En realidad, las versiones indirectas de las rutinas de colas de prioridad descritas en el Capítulo 11 son las más apropiadas para esta aplicación.) Específicamente, para hacer una fusión de P -vías se comienza llenando una cola de prioridad de tamaño P con el elemento más pequeño de cada una de las P entradas utilizando el procedimiento CP: :insertar del Capítulo 11 (adecuadamente modificado para que la raíz del montículo contenga al elemento más pequeño en lugar del más grande). Después, utilizando el procedimiento CP: :sustituir del Capítulo 11 (modificado de la misma manera) se da salida al elemento más pequeño y se reemplaza en la cola de prioridad por el siguiente elemento de su bloque.

El proceso de fusionar E E J con L M P y D E O (la primera fusión del ejemplo anterior) utilizando un montículo de tamaño tres se muestra en la Figura 13.3. Las «claves» de estos montículos son las más pequeñas (las primeras) de las claves de cada nodo. Por claridad, se muestran bloques enteros en los nodos del montículo; por supuesto, una implementación real consistiría en un montículo indirecto de punteros dentro de los bloques. Primero, se da salida a D, por lo que la E (la clave siguiente de su bloque) se convierte en la «clave» de la raíz. Como esto no viola la condición del montículo se da salida a la E, convirtiéndose O en la clave de la raíz. Esto sí viola la condición del montículo y por ello se intercambia el nodo con el que contiene E, E y J. A continuación se extrae la E y se sustituye por la siguiente clave de su bloque, la E. Esto no viola la condición del montículo, por lo que no es necesario ningún cambio más. Continuando de esta manera, se obtiene el archivo ordenado (leyendo la clave más pequeña del nodo raíz de los árboles de la Figura 13.3, para ver las claves en el orden en el que aparecen en la primera posición del montículo y en el que se

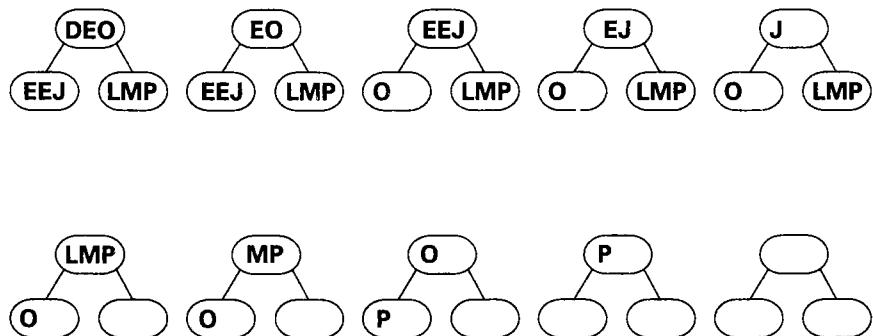


Figura 13.3 Selección por sustitución para la fusión sobre un montículo de tamaño tres.

obtienen en la salida). Cuando se agota un bloque, se pone un centinela en el montículo al que se considera mayor que todas las otras claves. Cuando el montículo no contiene más que centinelas, se ha terminado la fusión. A esta forma de utilizar las colas de prioridad se la denomina algunas veces *selección por sustitución*.

Así pues, para hacer una fusión de P -vías, se puede utilizar una selección por sustitución sobre una cola de prioridad de tamaño P para encontrar cada elemento a dar como salida en $\log P$ pasos. Esta diferencia de rendimiento no tiene ninguna repercusión práctica en particular, puesto que una implementación de fuerza bruta puede encontrar, en P pasos, cada elemento a dar como salida y P es normalmente tan pequeño que este coste es minúsculo en comparación con el de estar realmente dando salida al elemento. La importancia real de la selección por sustitución reside en la forma en que se puede utilizar en la primera parte del proceso de ordenación-fusión: formar los bloques iniciales ordenados que constituirán la base de las pasadas de la fusión.

La idea es pasar la entrada (desordenada) a través de una gran cola de prioridad, escribiendo siempre en la salida el elemento más pequeño de la cola, como antes, y sustituyéndolo por el siguiente elemento de la entrada, con un requisito adicional: si el nuevo elemento es menor que el último en salir, entonces, puesto que probablemente no podría formar parte del bloque que se está ordenando, se debería marcar como miembro del bloque siguiente y considerarse como superior a todos los elementos del bloque actual. Cuando un elemento marcado alcanza la cabeza de la cola de prioridad, se abandona el antiguo bloque y se comienza con uno nuevo. Una vez más, esto se implementa fácilmente con `CP::insertar` y `CP::sustituir` del Capítulo 11, apropiadamente modificados de modo que el elemento más pequeño esté en la raíz del montículo y cambiando `CP::sustituir` para que considere que los elementos marcados son siempre mayores que los no marcados.

El archivo ejemplo demuestra claramente el valor de la selección por susti-

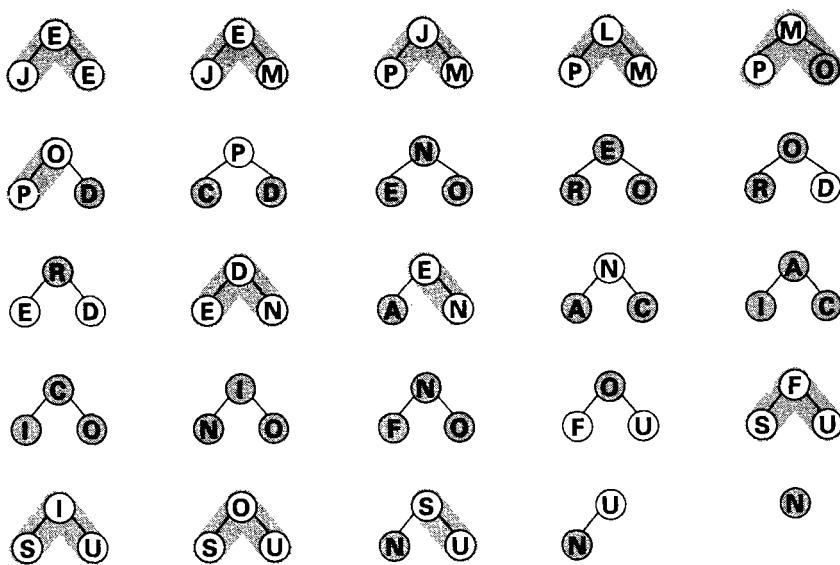


Figura 13.4 Creación de las secuencias iniciales de la selección por sustitución.

tución. Con una memoria interna capaz de contener sólo tres registros, se pueden producir bloques ordenados de tamaño 7, 4, 3, 5, 5 y 1, como se ilustra en la Figura 13.4. Como antes, el orden en que las claves ocupan las primeras posiciones del montículo es en el que se obtendrán en la salida. El sombreado indica a qué bloque pertenece cada clave del montículo: un elemento marcado de la misma forma que el de la raíz pertenece al bloque que está siendo ordenado y los otros pertenecen al bloque siguiente. La condición del montículo (la primera clave es menor que la segunda y la tercera) se mantiene en todas partes; los elementos del bloque siguiente se consideran como más grandes que los elementos del bloque que se está ordenando. La primera secuencia termina con D E O en el montículo, puesto que al llegar cada una de estas tres claves son más grandes que la raíz (por tanto no se pueden incluir en el primer bloque), la segunda termina con D E N, etcétera.

Propiedad 13.1 *Para claves aleatorias, las secuencias creadas por la selección por sustitución son aproximadamente del doble del tamaño del montículo utilizado.*

La demostración de esta propiedad necesita de hecho un análisis un poco más sofisticado, pero es fácil de verificar experimentalmente.■

El efecto práctico de esta propiedad es ganar una pasada de fusión: en vez de comenzar con secuencias ordenadas de aproximadamente el tamaño de la

memoria interna y después hacer una pasada de fusión para producir secuencias de alrededor del doble del tamaño de dicha memoria, se puede comenzar directamente con secuencias cuyo tamaño sea de unas dos veces el de la memoria interna, utilizando la selección por sustitución con una cola de prioridad de tamaño M . Si hay algún orden en las claves, entonces las secuencias serán mucho, mucho más largas. Por ejemplo, si ninguna clave tiene delante de ella en el archivo más de M claves superiores, ¡el archivo estará completamente ordenado después de la pasada de la selección por sustitución, y no será necesaria ninguna fusión! Ésta es la razón práctica más importante para utilizar el método.

En resumen, la técnica de selección por sustitución puede utilizarse a la vez para los pasos de «ordenación» y de «fusión» de una fusión múltiple equilibrada.

Propiedad 13.2 *Un archivo de N registros se puede ordenar utilizando una memoria interna capaz de contener M registros y con $P + 1$ cintas en alrededor de $1 + \log_p(N/2M)$ pasadas.*

Como se presentó anteriormente, se utiliza primero una selección por sustitución con una cola de prioridad de tamaño M , para producir secuencias iniciales de tamaño próximo a $2M$ (en una situación aleatoria) o más (si el archivo está parcialmente ordenado), y luego se utiliza la selección por sustitución con una cola de prioridad de tamaño P , para alrededor de $\log_p(N/2M)$ (o menos) pasadas de fusión. ■

Consideraciones prácticas

Para terminar de implementar el método antes esbozado, es necesario hacerlo con las funciones de entrada-salida que realmente transfieren los datos entre el procesador y los dispositivos externos. Estas funciones son evidentemente la clave del buen rendimiento de una ordenación externa, y necesitan que se consideren cuidadosamente (al contrario que los algoritmos) algunos aspectos del sistema. (Los lectores que no tengan ninguna relación con las computadoras a nivel de «sistema» pueden saltarse los próximos párrafos.)

Uno de los objetivos principales de la implementación debería ser el permitir un recubrimiento de la lectura, la escritura y los cálculos, tanto como sea posible. La mayoría de los grandes sistemas informáticos tienen unidades de procesamiento independientes para el control de los dispositivos de entrada/salida (E/S) en gran escala, lo que hace posible este recubrimiento. La eficacia que se puede alcanzar con un método de ordenación externa depende del número de dispositivos de este tipo.

Para cada archivo que se está leyendo o escribiendo, se puede utilizar la técnica de programación de sistemas denominada *doble buffer* para hacer máximo

el recubrimiento de E/S con el cálculo. La idea es mantener dos «buffers», uno reservado para el procesador principal y el otro para el dispositivo de E/S (o del procesador que controla al dispositivo de E/S). Para la entrada, el procesador utiliza un buffer mientras el dispositivo de entrada está llenando el otro. Cuando el procesador termina de utilizar su buffer, espera hasta que el dispositivo de entrada llene el suyo, y entonces los buffers intercambian sus papeles: el procesador utiliza los datos del buffer que se acaba de llenar mientras que el dispositivo de entrada vuelve a llenar el buffer que tenía los datos que el procesador acaba de utilizar. La misma técnica se utiliza para la salida, cambiando los papeles del procesador y el dispositivo. Habitualmente el tiempo de E/S es mucho más grande que el de procesamiento y, por lo tanto, el efecto del doble buffer es recubrir totalmente el tiempo de cálculo; por consiguiente los buffers deben ser tan grandes como sea posible.

Una dificultad del doble buffer es que realmente utiliza sólo la mitad del espacio de memoria disponible. Esto puede conducir a una falta de eficacia si existen muchos buffers, como es el caso de la fusión de P -vías cuando P no es pequeño. Este problema se puede soslayar utilizando una técnica denominada *previsión*, que necesita sólo un buffer extra (y no P) durante el proceso de fusión. La previsión funciona de la siguiente forma: ciertamente la mejor forma de recubrir la entrada con los cálculos durante el proceso de selección por sustitución es recubrir la entrada del siguiente buffer que se necesita llenar con la parte de procesamiento del algoritmo. Y es fácil determinar qué buffer es éste: el siguiente buffer de entrada a vaciar es aquel cuyo *último* elemento es el más pequeño. Por ejemplo, cuando se fusiona E E J con L M P y D E O se sabe que el primer buffer será el primero a vaciar, luego es el primero. Una forma simple de recubrir el procesamiento con la entrada en una fusión múltiple consiste, por lo tanto, en conservar un buffer extra que se llenará por el dispositivo de entrada de acuerdo con esta regla. Cuando el procesador encuentra un buffer vacío, espera hasta que el buffer de entrada esté lleno (si no se ha llenado ya), y luego cambia, para comenzar a utilizar este buffer, en lugar del vacío, al que dirige al dispositivo de entrada para que lo llene de nuevo de acuerdo con la regla de previsión.

La decisión más importante a tomar en la implementación de la fusión múltiple es la elección del valor de P , el «orden» de la fusión. Para ordenación en cinta, donde sólo se permite acceso secuencial, esta elección es fácil: P debe ser igual al número de unidades de cinta disponibles menos uno, puesto que la fusión múltiple utiliza P cintas de entrada y una de salida. Evidentemente, se deben tener al menos dos cintas de entrada, por tanto no tiene sentido tratar de ordenar en cintas si se dispone de menos de tres de ellas.

Para ordenación en disco, donde se permite el acceso a una posición arbitraria pero con un coste algo más caro que el acceso secuencial, es razonable escoger P igual al número de discos disponibles menos uno, para evitar el coste más elevado del acceso no secuencial que se produciría, por ejemplo, si dos archivos de entrada diferentes estuvieran en el mismo disco. Otra alternativa comúnmente utilizada es escoger P lo suficientemente grande para que la or-

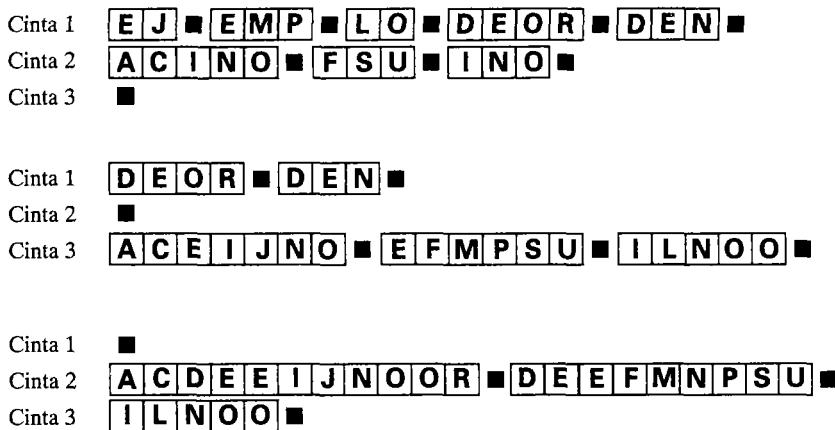
denación se complete en dos fases de fusión: normalmente no es razonable tratar de hacer la ordenación en una pasada, pero a veces se puede hacer en dos, con un P razonablemente pequeño. Puesto que la selección por sustitución produce alrededor de $N/2M$ secuencias y cada paso de fusión divide el número de secuencias por P , esto significa que el valor de P debe ser el menor entero tal que $P^2 > N/2M$. Para el ejemplo de ordenación de un archivo de 200 millones de palabras en una computadora con un millón de palabras de memoria, esto significa que $P = 11$ sería una buena elección para una ordenación de dos pasadas. (El valor exacto de P podría calcularse después de completar la fase de ordenación.) La mejor elección entre estas dos alternativas, del valor razonablemente más bajo de P y el valor razonablemente más alto de P , depende fuertemente de muchos parámetros del sistema: se deben considerar ambas alternativas (e incluso algunas intermedias).

Fusión polifásica

Uno de los problemas de la fusión múltiple equilibrada en la ordenación en cinta es que necesita o un número excesivo de unidades de cinta o una cantidad excesiva de copias. Para una fusión de P -vías o se utilizan $2P$ cintas (P para la entrada y P para la salida) o se debe copiar casi todo el archivo desde una cinta de salida a P cintas de entrada entre pasadas de fusión, lo que efectivamente dobla el número de pasadas, es decir, alrededor de $2\log_p(N/2M)$. Se han inventado varios algoritmos ingeniosos de ordenación en cintas, que eliminan virtualmente todas estas copias cambiando la forma en la que todos estos pequeños bloques ordenados se fusionan entre sí. El más extendido de estos métodos es el denominado *fusión polifásica*.

La idea básica que subyace en la fusión polifásica es distribuir los bloques ordenados, mediante una selección por sustitución, de forma irregular entre las unidades de cinta disponibles (dejando una vacía) y aplicando posteriormente una estrategia de «fusión hasta el vaciado», después de la cual las cintas de salida y entrada intercambian sus papeles.

Por ejemplo, se supone que se tienen exactamente tres cintas, y se parte de la configuración inicial de bloques ordenados en las cintas que se muestran en la parte superior de la Figura 13.5. (Esto se obtiene al aplicar la selección por sustitución al archivo ejemplo con una memoria interna que sólo puede contener dos registros.) La cinta 3 que está inicialmente vacía es la cinta de salida para las primeras fusiones. Después de tres fusiones de dos vías desde las cintas 1 y 2 hacia la cinta 3, la segunda cinta se vacía, como se muestra en la mitad de la Figura 13.5. A continuación, después de dos fusiones de dos vías desde las cintas 1 y 3 hacia la cinta 2, la primera cinta se vacía, como se muestra en la parte inferior de la Figura 13.5. La ordenación se completa en dos pasos más. Primero, una fusión de dos vías desde las cintas 2 y 3 hacia la cinta 1 deja un archivo en la cinta 2 y un archivo en la cinta 1, y después una fusión de dos vías

**Figura 13.5** Etapas iniciales de una fusión polifásica con tres cintas.

desde las cintas 1 y 2 hacia la cinta 3 deja el archivo totalmente ordenado en la cinta 3.

Esta estrategia de «fusión hasta el vaciado» se puede generalizar para trabajar con un número arbitrario de cintas. La Figura 13.6 muestra cómo se pueden utilizar seis cintas para ordenar 497 datos iniciales. Si se comienza como se indica en la primera columna de la Figura 13.6, con la cinta 2 como cinta de salida, la cinta 1 con 61 datos, la cinta 3 con 120, etc., entonces, después de ejecutar una «fusión hasta el vaciado» de cinco vías, se tendrá la cinta 1 vacía, la cinta 2 con 61 datos, la cinta 3 con 59, etc., como se muestra en la segunda columna de la Figura 13.6. En este momento se puede rebobinar la cinta 2 y convertirla en una cinta de entrada, y rebobinar la cinta 1 y convertirla en la cinta de salida. Continuando de esta forma, se llega a tener el archivo totalmente ordenado en la cinta 1. La fusión se corta en muchas *fases* que no implican a todos los datos, pero que no implican ninguna copia directa.

Cinta 1	61	0	31	15	7	3	1	0	1
Cinta 2	0	61	30	14	6	2	0	1	0
Cinta 3	120	59	28	12	4	0	2	1	0
Cinta 4	116	55	24	8	0	4	2	1	0
Cinta 5	108	47	16	0	8	4	2	1	0
Cinta 6	92	31	0	16	8	4	2	1	0

Figura 13.6 Distribución de secuencias para una fusión polifásica de seis cintas.

La principal dificultad en la implementación de una fusión polifásica es la de determinar cómo distribuir los datos iniciales. No es difícil ver cómo construir la tabla a la inversa: se toma el mayor número de cada columna, se convierte a cero, y se añade a cada uno de los otros números para obtener la columna anterior. Esto conduce a definir la fusión de mayor orden, para la columna anterior, que podría generar la columna actual. Esta técnica funciona para un número cualquiera de cintas (al menos tres): los números que aparecen son «números de Fibonacci generalizados» que poseen muchas propiedades interesantes. Por supuesto, el número de secuencias iniciales puede no conocerse por adelantado, y es probable que no sea exactamente un número generalizado de Fibonacci. Por tanto se puede añadir un cierto número de secuencias «ficticias» para hacer que el número de secuencias iniciales sea exactamente el que se necesita para la tabla.

El análisis de la fusión polifásica es complicado e interesante, y proporciona resultados sorprendentes. Por ejemplo, revela que el mejor método para distribuir las secuencias ficticias entre las cintas implica utilizar más fases y más secuencias de lo que parecería necesario. La razón de esto es que algunas secuencias se utilizan en las fusiones con más frecuencia que otras.

Para implementar un método más eficaz de ordenación en cinta se deben considerar otros muchos factores. Uno de los más importantes, que no se ha considerado en el capítulo, es el tiempo que se tarda en rebobinar la cinta. Este punto ha sido objeto de estudios detallados y se han definido muchos métodos fascinantes. Sin embargo, como se mencionó con anterioridad, las ganancias que se obtienen con respecto al método de la fusión múltiple equilibrada son bastante limitadas. Incluso la fusión polifásica sólo es más eficaz que la fusión equilibrada para un P pequeño, y no sustancialmente. Para $P > 8$, la fusión equilibrada posiblemente se ejecute con más rapidez que la polifásica, y para un P más pequeño el efecto de la polifásica es prácticamente el reducir en dos el número de cintas (una fusión equilibrada con dos cintas extra se ejecutaría más rápidamente).

Un método más fácil

Muchos sistemas de computadoras modernos incluyen dispositivos de *memoria virtual* de gran capacidad que no deben pasarse por alto al implementar un método para ordenar archivos muy grandes. En un buen sistema de memoria virtual, el programador puede acceder a cantidades muy grandes de datos, dejando al sistema la responsabilidad de transferir los datos desde el soporte de almacenamiento externo al interno, cuando sea necesario. Esta estrategia descansa en el hecho de que muchos programas presentan una localización relativamente pequeña de sus referencias: cada referencia a la memoria está en un área relativamente próxima a otra referenciada reciente-

mente. Esto implica que las transferencias desde la memoria externa a la interna son raramente frecuentes. Un método de ordenación interna con una pequeña localización de las referencias puede ser muy eficaz en un sistema de memoria virtual. (Por ejemplo, el Quicksort tiene dos «localizaciones»: la mayoría de las referencias están cerca de uno de los dos punteros de partición.) Pero es mejor recabar información de un programador de sistemas antes que estar esperando ganancias significativas: un método como el de ordenación por residuos, que no tiene ninguna localización de referencias, e incluso el Quicksort, podría provocar serios desastres en un sistema de memoria virtual, dependiendo de la forma de implementar el sistema de memoria virtual disponible. Por el contrario, la estrategia de utilizar un método simple de ordenación interna para ordenar archivos en disco merece una seria reflexión cuando se dispone de un buen sistema de memoria virtual.

Ejercicios

1. Describir cómo efectuaría el lector una *selección* externa: encontrar el K -ésimo elemento más grande de un archivo de N elementos, donde N es demasiado grande para que el archivo se pueda tener en la memoria interna.
2. Implementar el algoritmo de selección por sustitución y utilizarlo después para verificar la afirmación de que las secuencias generadas son aproximadamente del doble del tamaño de la memoria interna.
3. ¿Qué es lo *peor* que puede pasar cuando se utiliza la selección por sustitución para generar las secuencias iniciales en un archivo de N registros, utilizando una cola de prioridad de tamaño M , con $M < N$?
4. ¿Cómo se ordenaría el contenido de un disco si no existe otro medio de almacenamiento disponible que el de la memoria principal?
5. ¿Cómo se ordenaría el contenido de un disco si sólo hay disponible una sola cinta (y la memoria principal)?
6. Comparar la fusión múltiple equilibrada de cuatro y seis cintas con la fusión polifásica con el mismo número de cintas y 31 secuencias iniciales.
7. ¿Cuántas fases utiliza la fusión polifásica de cinco cintas cuando comienza con cuatro cintas que contienen inicialmente 26, 15, 22 y 28 secuencias?
8. Suponiendo que las 31 secuencias iniciales de una fusión polifásica de cuatro cintas tienen cada una la longitud de un registro (con la distribución inicial 0, 13, 11, 7), ¿cuántos registros hay en cada uno de los archivos implicados en la última fusión de tres vías?
9. ¿Cómo se deberían tratar los archivos pequeños en una implementación del Quicksort destinada a aplicarse en archivos muy grandes en un entorno de memoria virtual?

- 10.** ¿Cómo se organizaría una cola de prioridad externa? (Concretamente, diseñar una forma de soportar las operaciones *insertar* y *suprimir* del Capítulo 11, cuando el número de elementos de la cola de prioridad podría crecer de modo tal que fuera demasiado grande para mantenerla en la memoria principal.)

REFERENCIAS para la Ordenación

La referencia principal para esta sección es el Volumen 3 de la obra de D. E. Knuth, sobre ordenación y búsqueda. En este libro se puede encontrar información adicional sobre prácticamente todos los temas presentados con anterioridad. En particular, los resultados presentados aquí sobre las características del rendimiento de los diferentes algoritmos están respaldados por análisis matemáticos completos.

Existe una vasta literatura sobre ordenación. La bibliografía de Knuth y Rivest de 1973 contiene cientos de citas, pero no incluye el tratamiento de la ordenación que figura en innumerables libros y artículos sobre otros temas. El libro de Gonnet es una referencia más actualizada, que contiene una extensa bibliografía que cubre los trabajos hasta 1984.

Para el Quicksort, la mejor referencia es el artículo original de Hoare de 1962, que describe las variantes más importantes, incluyendo la utilización para el problema de la selección presentado en el Capítulo 9. Muchos más detalles sobre el análisis matemático y los efectos prácticos de diversas modificaciones y mejoras propuestas a lo largo de los años se pueden encontrar en el libro publicado en 1978 por el autor de esta obra.

Un buen ejemplo de una estructura avanzada de cola de prioridad es la «cola binomial» de J. Vuillemin, implementada y analizada por M. R. Brown. Esta estructura de datos permite todas las operaciones de cola de prioridad de una forma elegante y eficaz. El tipo de estructura de datos más avanzado para implementaciones prácticas es el «montículo pareado», descrito por Fredman, Sedgewick, Sleator y Tarjan.

Para tener una impresión sobre la infinidad de detalles relativos a la transposición de algoritmos como los que se han presentado en implementaciones prácticas de uso general, se sugiere al lector que estudie los manuales de referencia de los sistemas de ordenación de su computadora. Estos manuales hacen necesariamente una revisión de los formatos de las claves, registros y archivos, así como de otros detalles, y a menudo es interesante constatar cómo entran en juego los propios algoritmos.

- M. R. Brown, «Implementation and analysis of binomial queue algorithms», *SIAM Journal of Computing*, 7, 3 (agosto, 1978).
- M. L. Fredman, R. Sedgewick, D. D. Sleator y R. E. Tarjan, «The pairing heap: a new form of self-adjusting heap», *Algorithmica*, 1, 1 (1986).
- G. H. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1984.
- C. A. R. Hoare, «Quicksort», *Computer Journal*, 5, 1 (1962).
- D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, segunda impresión, Addison-Wesley, Reading, MA, 1975.
- R. L. Rivest y D. E. Knuth, «Bibliography 26: Computing Sorting», *Computing Reviews*, 13, 6 (junio, 1972).
- R. Sedgewick, *Quicksort*, Garland, New York, 1978. (Aparece también como tesis de Ph.D., Universidad de Stanford, 1975.)

Algoritmos de búsqueda

Métodos de búsqueda elementales

La *búsqueda* es una operación fundamental, intrínseca a una gran cantidad de tareas de las computadoras, que consiste en recuperar uno o varios elementos particulares de un gran volumen de información previamente almacenada. Normalmente se considera que la información está dividida en *registros*, cada uno de los cuales posee una clave para utilizar en la búsqueda. El objetivo de esta operación es encontrar todos los registros cuyas claves coincidan con una cierta *clave de búsqueda*, con el propósito de acceder a la información (y no solamente a la clave) para su procesamiento.

Las aplicaciones de la búsqueda están muy difundidas y abarcan una variada gama de operaciones diferentes. Por ejemplo, un banco necesita hacer un seguimiento de las cuentas de todos sus clientes y buscar en ellas para verificar diversos tipos de transacciones. De igual forma un sistema de reservas de unas líneas aéreas tiene necesidades similares, aunque la mayor parte de los datos sean de vida corta.

Dos términos comunes que se utilizan a menudo para describir las estructuras de datos relativas a las búsquedas son los *diccionarios* y las *tablas de símbolos*. Por ejemplo, en un diccionario de inglés las «claves» son las palabras y los «registros» las entradas asociadas con ellas, que contienen la definición, la pronunciación y otras informaciones. Se puede aprender un método de búsqueda y apreciar su acción pensando cómo se implementaría un sistema para buscar en un diccionario de inglés. Una tabla de símbolos es el diccionario de un programa: las «claves» son los nombres simbólicos utilizados en el programa y los «registros» contienen la información que describe al objeto designado.

En la búsqueda (como en la ordenación) existen programas que están muy difundidos y se utilizan frecuentemente, de modo que merece la pena estudiar con cierto detalle un cierto número de métodos. Al igual que en la ordenación, se comenzará por estudiar algunos métodos elementales, que son muy útiles en

pequeñas tablas y en otras situaciones especiales, y después se mostrarán las técnicas fundamentales a explotar por los métodos más avanzados. Se verán métodos que almacenan los registros en arrays, en los que se busca por comparación entre claves o que están indexados por el valor de la clave, y posteriormente se verá un método fundamental que construye estructuras definidas por los valores de las claves.

Al igual que en las colas de prioridad, es preferible considerar que los algoritmos de búsqueda pertenecen a conjuntos de rutinas empaquetadas que realizan una serie de operaciones genéricas y que se pueden disociar de las implementaciones particulares, de tal forma que permiten pasar fácilmente de una implementación a otra. Entre las operaciones que interesan se cuentan:

Iniciar la estructura de datos.

Buscar un registro (o varios) con una clave dada.

Insertar un nuevo registro.

Eliminar un registro específico.

Unir dos diccionarios en uno solo (de mayor tamaño).

Ordenar el diccionario; dar como salida todos los registros ordenados.

Al igual que en las colas de prioridad, a veces es conveniente combinar algunas de estas operaciones. Por ejemplo, la operación *buscar e insertar* se incluye a menudo, por razones de eficacia, en situaciones en las que la estructura de datos no debe contener registros con claves iguales. En muchos métodos, una vez que se ha determinado que una clave no pertenece a la estructura de datos, el propio estado interno del procedimiento de búsqueda contiene la información necesaria para insertar un nuevo registro con la clave dada.

Los registros con claves iguales se pueden tratar de varias formas, según la aplicación. Primero, se puede insistir para que la estructura de datos primaria contenga sólo registros con claves distintas. Entonces cada «registro» de esta estructura de datos puede contener, por ejemplo, una lista enlazada de todos los registros que tienen la misma clave. Esto es conveniente en algunas aplicaciones, puesto que *todos* los registros con la misma clave se obtendrán en una sola *búsqueda*. Una segunda posibilidad es colocar a todos los registros con la misma clave en la estructura de datos primaria y devolver, en una *búsqueda*, *cualquier* registro que contenga la clave. Esto es más simple en aplicaciones que procesan registro a registro, donde no es importante el orden en el que se procesan los registros que tienen claves iguales. Esta solución no es satisfactoria para el diseño de un algoritmo porque se debe proporcionar un mecanismo para recuperar *otro* registro o *todos* los registros con la misma clave. Una tercera posibilidad consiste en suponer que cada registro tiene un identificador único (aparte de la clave) y entonces la *búsqueda* ha de encontrar el registro que tiene el identificador dado, conociendo la clave. Una cuarta posibilidad es que el programa de búsqueda llame a una función específica para cada registro que tenga la clave dada. También podrían ser necesarios otros mecanismos más complejos. En este libro, al describir los algoritmos de búsqueda, se menciona informalmente cómo

se pueden encontrar registros con claves iguales, sin precisar qué mecanismo hay que utilizar. Los ejemplos del capítulo contendrán normalmente claves iguales.

Cada una de las operaciones fundamentales antes enunciadas tiene aplicaciones importantes y se han sugerido un gran número de organizaciones básicas que permiten el uso eficaz de diversas combinaciones de ellas. En éste y en los próximos capítulos, se centrará la atención en las implementaciones de las funciones fundamentales de *buscar* e *insertar* (y por supuesto *inicializar*), con algunos comentarios sobre *eliminar* y *ordenar* cuando sea conveniente. Al igual que en las colas de prioridad, la operación *unión* necesita normalmente técnicas que se salen del marco de este tratamiento.

Búsqueda secuencial

El método de búsqueda más simple consiste en almacenar todos los registros en un array. Cuando se inserta un nuevo registro, se pone al final del array; cuando se lleva a cabo una búsqueda, se recorre secuencialmente el array. El siguiente programa muestra una implementación de las funciones básicas que utiliza esta sencilla organización e ilustra a su vez algunos de los convenios que se utilizarán en la implementación de los métodos de búsqueda.

```
class Dicc
{
    private:
        struct nodo
        { tipoElemento clave; tipoInfo info; };
        struct nodo *a;
        int N;
    public:
        Dicc(int max)
        { a = new nodo[max]; N = 0; }
        ~Dicc()
        { delete a; }
        tipoInfo buscar(tipoElemento v);
        void insertar(tipoElemento v, tipoInfo info);
    };
    tipoInfo Dicc::buscar(tipoElemento v) // Secuencial
    {
        int x = N+1;
        a[0].clave = v; a[0].info = infoNIL;
        while (v != a[--x].clave) ;
        return a[x].info;
    }
}
```

```

    }
    void Dicc::insertar(tipoElemento v, tipoInfo info)
    { a[++N].clave = v; a[N].info = info; }
```

Ésta es una implementación de un tipo de datos de diccionario donde las claves (`clave`) se utilizan para almacenar y recuperar la «información asociada» (`info`). Al igual que en la ordenación, a veces será necesario ampliar los programas para manipular registros y claves más complicadas, pero esto no implica cambios fundamentales en los algoritmos. Por ejemplo, si `tipoElemento` fuera `char*` y se sobrecargara el operador `!=` para hacer `strcmp` convertiría al programa anterior en un paquete que utilizaría como claves a cadenas de caracteres en lugar de enteros. O `info` podría ser un puntero a una estructura de registro más compleja. Así, este campo puede servir como identificador único del registro para distinguir entre registros con claves iguales.

Aquí, buscar devuelve el campo `info` del primer registro encontrado que tenga la clave en cuestión (`infoNIL` si no existe tal registro).

Se utiliza un registro centinela en el que su campo `clave` se inicializa con el valor a buscar para garantizar que la búsqueda siempre terminará y por lo tanto el bucle interno se podrá escribir con solamente una comprobación de terminación. Al campo `info` de este registro centinela se le asigna el valor `infoNIL` de manera que sea éste el valor devuelto cuando ningún registro tenga la clave dada. Esta técnica es análoga a la del registro centinela que contiene el valor máximo o mínimo de una clave, que se utiliza para simplificar la escritura de varios algoritmos de ordenación.

Propiedad 14.1 *La búsqueda secuencial (implementación por array) utiliza (siempre) $N + 1$ comparaciones para una búsqueda sin éxito y alrededor de $N/2$ comparaciones (por término medio) para una búsqueda con éxito.*

Para una búsqueda sin éxito, esta propiedad se deduce directamente del programa: se debe examinar cada registro para decidir si una clave en particular está ausente. Para una búsqueda con éxito, si se supone que todos los registros tienen la misma probabilidad de ser el buscado, el número medio de comparaciones es $(1 + 2 + \dots + N)/N = (N + 1)/2$, exactamente la mitad del coste de una búsqueda infructuosa.■

Es obvio que la búsqueda secuencial se puede adaptar de manera natural para utilizar una representación de los registros mediante una lista enlazada:

```

class Dicc
{
private:
    struct nodo
    { tipoElemento clave; tipoInfo info;
      struct nodo *siguiente;
```

```

nodo(tipoElemento k, tipoInfo i, struct nodo *n)
    { clave = k; info = i; siguiente = n; };
};

struct nodo *cabeza, *z;
public:
    Dicc(int max)
    {
        z = new nodo(elementoMAX, infoNIL, 0);
        cabeza = new nodo(0, 0, z);
    }
    ~Dicc();
    tipoInfo buscar(tipoElemento v);
    void insertar(tipoElemento v, tipoInfo info);
};

```

Como es habitual en las listas enlazadas, un nodo cabecera ficticio cabeza y un nodo cola z permiten simplificar el código. Se ha pasado al estilo de utilizar un constructor para hacer más conveniente la operación de llenar los campos de los nodos a la vez que se van creando. La búsqueda implica un trabajo más creativo que el desarrollado en los primeros capítulos.

Una razón para utilizar una lista enlazada es que es fácil mantener la lista ordenada (se verá posteriormente). Esto hace la búsqueda más eficaz: puesto que la lista está ordenada, cada búsqueda puede terminar cuando se encuentre un registro con una clave no menor que la clave de búsqueda.

```

tipoInfo Dicc::buscar(tipoElemento v) //Lista ordenada
{
    struct nodo *t = cabeza;
    while (v > t->clave) t = t->siguiente;
    return (v = t->clave) ? t->info : z->info;
}

```

Es fácil mantener el orden insertando cada nuevo registro en el lugar donde termina la búsqueda sin éxito:

```

void Dicc::insertar(tipoElemento v, tipoInfo info)
{
    struct nodo *x, *t = cabeza;
    while (v > t->siguiente->clave) t = t->siguiente;
    x = new nodo(v, info, t->siguiente);
    t->siguiente = x;
}

```

Este programa es una implementación alternativa del mismo tipo de datos abstracto de la implementación por array anterior. Las dos versiones permiten la inserción, la búsqueda y la inicialización. Se continuará la programación de algoritmos de búsqueda de esta manera, añadiendo otras funciones cuando sea apropiado. Por otra parte, las implementaciones podrán utilizarse de forma equivalente en las aplicaciones, diferenciándose solamente (es de esperar) en las necesidades de tiempo y espacio. Por ejemplo, sería trivial añadir una función ordenar a esta implementación por lista enlazada, pero para añadir ordenar a la implementación por array anterior habría que reprogramar alguno de los métodos de los capítulos 8 al 12.

Propiedad 14.2 *Una búsqueda secuencial (en una implementación por lista ordenada) utiliza alrededor de $N/2$ comparaciones (por término medio) para las dos búsquedas (con éxito o sin él).*

Para la búsqueda con éxito, la situación es la misma que antes. Para la búsqueda sin éxito, si se supone que existe la misma probabilidad de que la búsqueda acabe en el nodo terminal z o en cualquiera de los elementos de la lista (que es el caso de un cierto número de modelos de búsqueda «aleatoria»), entonces el número medio de comparaciones es el mismo que el de una búsqueda con éxito en una tabla de tamaño $N + 1$, o sea $(N + 2)/2$. ■

Se podría también desarrollar fácilmente una implementación por «lista desordenada» para la búsqueda secuencial, con características similares a la de la implementación por array. Por ejemplo, si las búsquedas son relativamente poco frecuentes, entonces el tiempo constante de la *inserción* puede ser una ventaja.

Si se conoce algo sobre la frecuencia relativa de acceso de diferentes registros, se pueden lograr mejoras sustanciales simplemente ordenando los registros inteligentemente. La ubicación «óptima» consiste en poner el registro de acceso más frecuente en el comienzo, el segundo de acceso más frecuente en la segunda posición, etc. Esta técnica puede ser muy eficaz, en especial si sólo se consulta frecuentemente un pequeño conjunto de registros.

Si no hay información disponible sobre la frecuencia de acceso, entonces se puede lograr una aproximación a la ubicación óptima con una búsqueda «autoorganizada»: cada vez que se acceda a un registro se le coloca al principio de la lista. Este método es más conveniente de implementar cuando se utiliza una lista enlazada. Por supuesto el tiempo de ejecución depende de las distribuciones de acceso a los registros; así pues, es difícil predecir el comportamiento general del método. Pero esto es eficaz en la situación usual en la que se accede de manera repetitiva a muchos registros que están próximos entre sí.

Búsqueda binaria

Si el conjunto de registros es grande, entonces el tiempo total de búsqueda se puede reducir significativamente utilizando un procedimiento de búsqueda ba-

sado en la aplicación del paradigma de «divide y vencerás»: se divide el conjunto de registros en dos partes, se determina a cuál de las dos partes debe pertenecer la clave buscada, y a continuación se repite el proceso en esa parte. Una forma razonable de dividir en partes el conjunto de registros consiste en mantener los registros ordenados y después utilizar los índices del array ordenado para delimitar la parte del array sobre la que se va a trabajar:

```

tipoInfo Dicc::buscar(tipoElemento v) //Búsqueda binaria
{
    int izq = 1; int der = N; int x;
    while (der >= izq)
    {
        x = (izq + der)/2;
        if (v == a[x].clave) return a[x].info;
        if (v < a[x].clave) der = x-izq; else izq = x+izq;
    };
    return infoNIL;
}

```

Para averiguar si una clave dada v está en la tabla, primero se le compara con el elemento de la posición intermedia de la tabla. Si v es menor, entonces debe estar en la primera mitad de la tabla; si v es mayor, entonces debe estar en la segunda mitad de la tabla. A continuación se aplica esta técnica recursivamente. Puesto que sólo interviene una llamada recursiva, es más simple expresar el método iterativamente.

Al igual que en el Quicksort y la ordenación por intercambio de residuos, este método utiliza los punteros izq y der para delimitar el subarchivo sobre el que se está trabajando. Si este subarchivo llega a estar vacío, entonces la búsqueda resultará infructuosa. En otro caso la variable x se fija con el valor del punto medio del intervalo, existiendo tres posibilidades: o se encuentra un registro con la clave dada, o bien el puntero izquierdo se cambia a $x+izq$, o bien el puntero derecho se cambia a $x-izq$, según que el valor v buscado sea igual, menor o mayor que el valor de la clave del registro almacenado en $a[x]$.

La Figura 14.1 muestra los subarchivos examinados por este método cuando se busca O en una tabla construida insertando las claves E J E M P L O D E B U S Q U E D A. El tamaño del intervalo se reduce a la mitad en cada paso, de tal modo que sólo se utilizan cuatro comparaciones en la búsqueda. La Figura 14.2 muestra un ejemplo mayor, con 95 registros; aquí a lo sumo se requieren siete comparaciones para cualquier búsqueda.

Propiedad 14.3 *La búsqueda binaria nunca utiliza más de $\lg N + 1$ comparaciones para cada búsqueda (con éxito o sin él).*

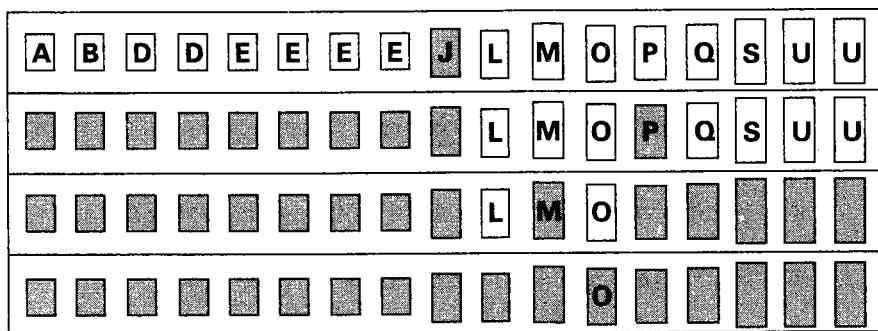


Figura 14.1 Búsqueda binaria.

Esto se deduce del hecho de que el tamaño del subarchivo se reduce al menos a la mitad en cada paso: una cota superior del número de comparaciones satisface la recurrencia $C_N = C_{N/2} + 1$ con $C_1 = 1$, lo que implica el resultado indicado (fórmula 2 del Capítulo 6).■

Es importante notar que en el caso de búsqueda binaria el tiempo que se necesita para *insertar* nuevos registros es elevado: el array debe mantenerse ordenado, de modo que algunos registros deberán moverse para dejar sitio a uno nuevo. Si un nuevo registro tiene una clave inferior a la de cualquier registro de la tabla, entonces cada uno de ellos debe moverse una posición. Una inserción aleatoria requiere que se muevan $N/2$ registros por término medio. Por ello este método no debe utilizarse en aplicaciones que impliquen muchas inserciones. Este método constituye la mejor elección en situaciones en las que la tabla se puede «construir» de una vez desde el principio, quizás por medio de un método de ordenación como el de Shell o el Quicksort, y utilizarse después para un gran número de búsquedas (muy eficaces).

La búsqueda con éxito para el *info* asociado con una clave *v*, presente múltiples veces, terminará en algún lugar dentro de un bloque contiguo de registros que tienen la clave *v*. Si la aplicación necesita acceder a todos estos registros, se pueden encontrar recorriendo ambas direcciones a partir del punto en el que se terminó la búsqueda. Una técnica similar se puede utilizar para resolver el problema más general de encontrar todos los registros cuyas claves están dentro de un determinado intervalo.

La secuencia de comparaciones realizadas por el algoritmo de búsqueda binaria es predeterminada: la secuencia específica depende del valor de la clave que se está buscando y del valor de *N*. La estructura de comparación se puede describir de forma sencilla mediante una estructura de árbol binario. La Figura 14.3 muestra la estructura de las comparaciones para el ejemplo anterior del conjunto de claves. Por ejemplo, al buscar un registro con la clave O, primero

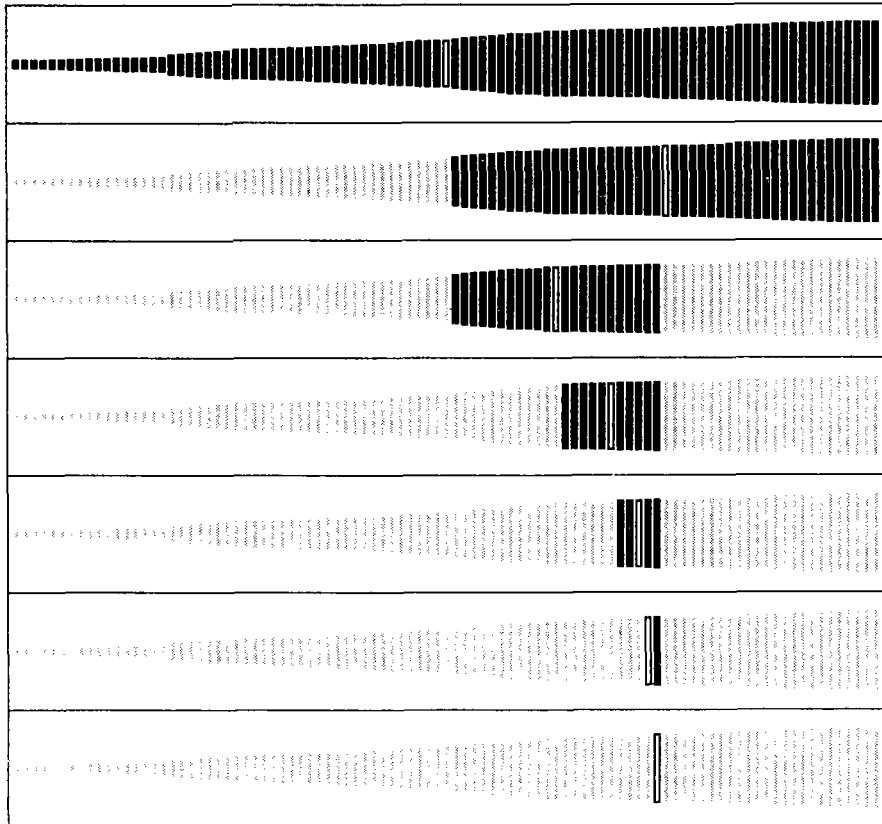


Figura 14.2 Búsqueda binaria en un archivo muy grande.

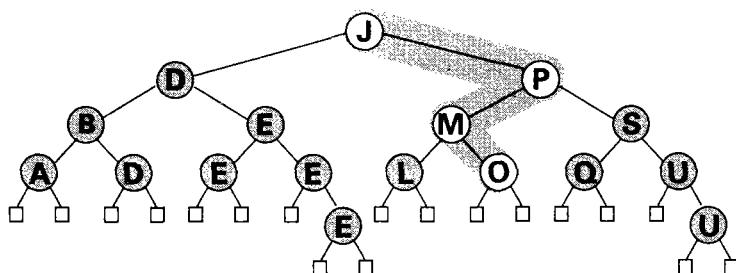


Figura 14.3 Árbol de comparaciones para la búsqueda binaria.

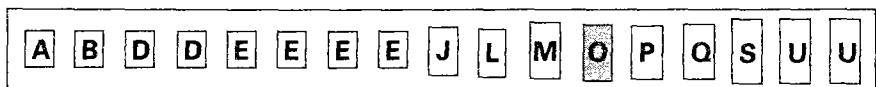


Figura 14.4 Búsqueda por interpolación.

se compara con J. Puesto que O es mayor, a continuación se compara con P (en el caso contrario se habría comparado con D), luego se compara con M y el algoritmo termina, con éxito, en la cuarta comparación. Más adelante se verán algoritmos que utilizan un árbol binario construido explícitamente para guiar la búsqueda.

Una posible mejora de la búsqueda binaria consiste en tratar de acertar en qué parte del intervalo está la clave que se está buscando (mejor que utilizar ciegamente la mitad del intervalo en cada paso). Esta técnica imita la forma como se busca un número en una guía telefónica. Por ejemplo: si el nombre comienza con B se mira cerca del principio, pero si comienza con Y, se mira cerca del final. Este método, denominado *búsqueda por interpolación*, requiere sólo una simple modificación del programa precedente. En él, el nuevo punto de partida de la búsqueda (el punto medio del intervalo) se calcula por medio de la sentencia $x = (\text{izq} + \text{der})/2$, que se deduce de la expresión

$$x = \text{izq} + \frac{1}{2}(\text{der} - \text{izq}).$$

El punto medio del intervalo se calcula añadiendo la mitad de su tamaño al punto izquierdo del mismo. En la búsqueda por interpolación simplemente se sustituye la fracción $1/2$ de esta fórmula por una estimación de donde puede encontrarse la clave, sobre la base de los valores disponibles: el $1/2$ sería apropiado si v estuviera en la mitad del intervalo entre $a[\text{izq}].\text{clave}$ y $a[\text{der}].\text{clave}$, pero $x = \text{izq} + (v - a[\text{izq}].\text{clave}) * (\text{der} - \text{izq}) / (a[\text{der}] . \text{clave} - a[\text{izq}] . \text{clave})$ podría ser una estimación mejor (si las claves son numéricas y están uniformemente distribuidas).

Suponiendo en el ejemplo que la i -ésima letra del alfabeto se representa por el número i . Entonces en la búsqueda de O, la primera posición a examinar en la tabla sería la 12, puesto que $1 + (15 - 1)*(17 - 1)/(21 - 1) = 12,2 \dots$ La búsqueda se completa en un solo paso (12 es la posición de O en la clave). Incluso tomando para la primera posición a examinar el valor encontrado por exceso (13; que corresponde a P), la búsqueda se completaría en dos pasos. El primer y el último elementos se localizan en el primer paso. La Figura 14.5 muestra la búsqueda por interpolación en el archivo de 95 elementos de la Figura 14.2; dicha búsqueda utiliza solamente cuatro comparaciones, cuando la binaria necesita siete.

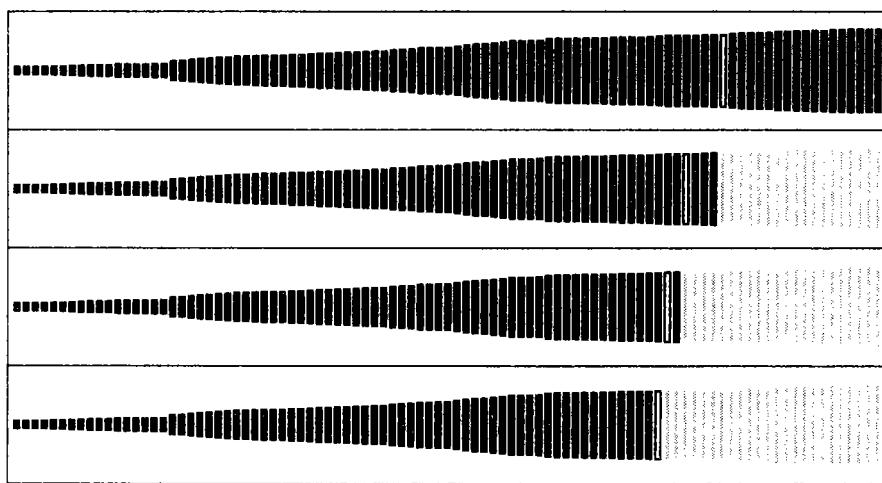


Figura 14.5 Búsqueda por interpolación en un gran archivo.

La búsqueda por interpolación utiliza menos de $\lg\lg N + 1$ comparaciones, lo mismo para una búsqueda con éxito que para una infructuosa, en archivos con claves aleatorias.

La demostración de este hecho rebasa el alcance de este libro. Esta función tiene un crecimiento muy lento que se puede considerar como una constante para propósitos prácticos: si N es mil millones, entonces $\lg\lg N < 5$. De este modo se puede encontrar cualquier registro utilizando sólo unos pocos accesos (por término medio), lo que representa una mejora sustancial con relación a la búsqueda binaria.■

Sin embargo, la búsqueda por interpolación depende fuertemente de la suposición de que las claves están bien distribuidas en el intervalo, por lo que a esta técnica la puede «engañar» una distribución poco uniforme de las claves, lo que es frecuente en la práctica. Además este método requiere algunos cálculos: para un N pequeño, el coste de $\lg N$ de la búsqueda binaria directa es bastante próximo a $\lg\lg N$, por lo que no merece la pena pagar tanto por la interpolación. Por el contrario, la búsqueda por interpolación debe tenerse en cuenta para el caso de archivos grandes, en aplicaciones donde las comparaciones son muy costosas o en métodos externos que implican costes muy altos.■

Búsqueda por árbol binario

La búsqueda por árbol binario es un método simple y eficaz de búsqueda dinámica, que está calificado como uno de los algoritmos fundamentales de la in-

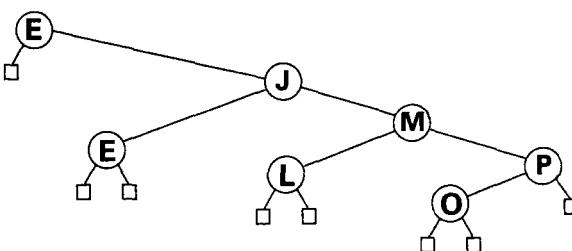


Figura 14.6 Un árbol binario de búsqueda.

formática. Se presenta entre los métodos «elementales» por lo simple que es; pero de hecho es el método elegido en muchas situaciones.

En el Capítulo 4 se estudiaron con detalle los árboles, y de él se recuerda ahora la terminología: la propiedad característica de un *árbol* es que sobre cada nodo apunta otro único nodo denominado su *padre*, y la propiedad característica de un *árbol binario* es que cada nodo tiene dos enlaces (apunta a dos nodos), uno izquierdo y otro derecho. Para su empleo en la búsqueda, cada nodo tiene también un registro con un valor clave. En un *árbol binario de búsqueda* se impone que todos los registros con las claves más pequeñas están en el subárbol izquierdo y que todos los registros del subárbol derecho tienen valores de clave mayores o iguales. Pronto se verá que es muy fácil garantizar que los árboles binarios de búsqueda construidos por inserciones sucesivas de nuevos nodos satisfagan también esta propiedad de definición. En la Figura 14.6 se muestra un ejemplo de árbol binario de búsqueda; como es ya usual, los subárboles vacíos se representan por pequeños nodos cuadrados.

De esta estructura se desprende inmediatamente un procedimiento de búsqueda binaria. Para encontrar un registro con una clave dada v , primero se compara ésta con la correspondiente a la raíz. Si es más pequeña, se va al subárbol de la izquierda; si es igual, se detiene la búsqueda; si es mayor, se va al subárbol de la derecha. Se aplica este método recursivamente. En cada paso, se tiene la garantía de que ninguna otra parte del árbol que no sea la del subárbol en el que se está situado puede contener registros con la clave v , y, al igual que disminuye el tamaño del intervalo en la búsqueda binaria, el «subárbol actual» es cada vez más pequeño. El procedimiento termina cuando se encuentra un registro con clave v o, si no hay tal registro, cuando el «subárbol actual» llega a estar vacío. Llegados a este punto hay que admitir que las palabras «binaria», «búsqueda» y «árbol» están sobreutilizadas, y el lector debe estar seguro de comprender la diferencia entre la función de búsqueda binaria presentada anteriormente en este capítulo y los árboles binarios de búsqueda descritos aquí. En una búsqueda binaria, se utiliza un árbol binario para describir la secuencia de comparaciones llevada a cabo por una función que busca en un array; aquí realmente se construye una estructura de datos en forma de árbol, con registros conectados por enlaces, y se utiliza para la búsqueda.

```

class Dicc
{
    private:
        struct nodo
        { tipoElemento clave; tipoInfo info;
          struct nodo *izq, *der;
          nodo(tipoElemento k, tipoInfo i,
                struct nodo *izqizq, struct nodo *derder)
          { clave=k; info=i; izq= izqizq; der=derder; };
        };
        struct nodo *cabeza, *z;
    public:
        Dicc(int max)
        { z = new nodo(0, infoNIL, 0, 0);
          cabeza = new nodo(elementoMIN, 0, 0, z); }
        ~Dicc();
        tipoInfo buscar(tipoElemento v);
        void insertar(tipoElemento v, tipoInfo info);
    };
    tipoInfo Dicc::buscar(tipoElemento v)
    {
        struct nodo *x = cabeza->der;
        z->clave = v;
        while (v != x->clave)
            x = (v < x->clave) ? x->izq : x->der;
        return x->info;
    }
}

```

Es conveniente utilizar un nodo cabeza que sea la cabecera del árbol cuyo enlace derecho apunte al nodo raíz real del árbol y cuya clave sea inferior a todas las otras. El enlace izquierdo de cabeza no se utiliza. La utilidad de cabeza se verá más clara posteriormente, cuando se presente la inserción. Si un nodo no tiene subárbol izquierdo (derecho) entonces su enlace izquierdo (derecho) se pone a apuntar al nodo «final» z. Al igual que en la búsqueda secuencial, se coloca en z el valor que se busca, para detener las búsquedas infructuosas. Así, el «subárbol actual» sobre el que apunta x nunca estará vacío y todas las búsquedas tendrán «éxito»: la inicialización de z->info a infoNIL servirá para indicar, al devolver este indicador, que una búsqueda no ha tenido éxito de acuerdo con el convenio que se ha venido utilizando. Los programas de este capítulo nunca acceden a los enlaces de z, pero para los programas más avanzados que se verán más adelante es conveniente inicializar los enlaces de z para que apunten al propio z.

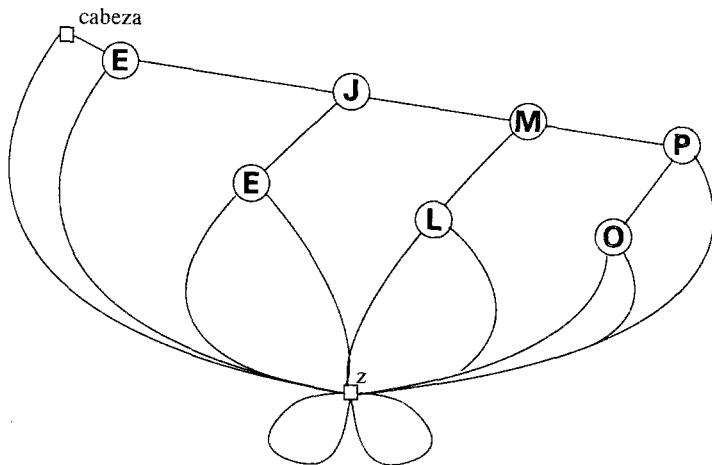


Figura 14.7 Un árbol binario de búsqueda (con nodos ficticios).

Como se mostró anteriormente, en la Figura 14.6, es conveniente representar los enlaces que apuntan a z como si apuntaran a unos *nodos externos* imaginarios, y que todas las búsquedas sin éxito terminan en nodos externos. Los nodos normales que contienen las claves se denominan *nodos internos*. Al introducir nodos externos se puede afirmar que todo nodo interno apunta a otros dos nodos del árbol, aun cuando en esta implementación todos los nodos externos estén representados por el único nodo z . La Figura 14.7 muestra explícitamente estos enlaces y los nodos ficticios.

La Figura 14.8 muestra lo que sucede cuando se busca D en el árbol ejemplo, utilizando buscar. Primero, se compara con E, la clave de la raíz. Puesto que D es menor, se va hacia la izquierda y, como el enlace izquierdo del nodo que contiene a E es un puntero a z , la búsqueda termina: D se compara consigo mismo en z y la búsqueda resulta infructuosa.

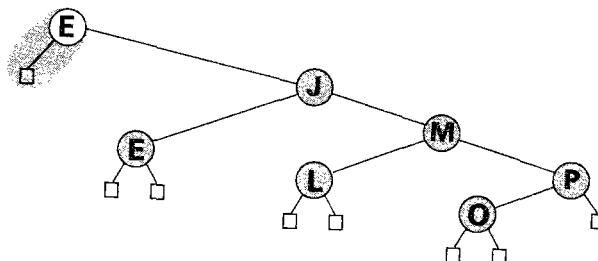


Figura 14.8 Búsqueda (de D) en un árbol binario de búsqueda.

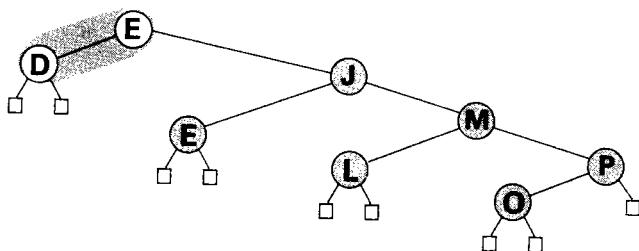


Figura 14.9 Inserción (de D) en un árbol binario de búsqueda.

Para insertar un nodo en el árbol, se efectúa una búsqueda infructuosa de su clave y a continuación se agrega el nuevo nodo en lugar de z en el punto donde se terminó la búsqueda. Para hacer la inserción, el siguiente programa sigue la pista del padre p de x a medida que se desciende por el árbol.

Cuando se alcanza el fondo del árbol ($x == z$), p apunta al nodo cuyo enlace debe cambiarse para apuntar al nuevo nodo insertado.

```

void Dicc::insertar(tipoElemento v, tipoInfo info)
{
    struct nodo *p, *x;
    p = cabeza; x = cabeza->der;
    while (x != z)
        { p = x; x = (v < x->clave) ? x->izq : x->der; };
    x = new nodo(v, info, z, z);
    if (v < p->clave) p->izq = x; else p->der = x;
}
  
```

En esta implementación, cuando se inserta un nuevo nodo cuya clave es igual a alguna de las que ya existen en el árbol, se insertará a la derecha del nodo que ya estaba en el árbol. Esto significa que se pueden encontrar los nodos con claves iguales si simplemente se continúa la búsqueda a partir del punto en el que buscar terminó, hasta que se encuentre z.

El árbol de la Figura 14.9 se obtiene al insertar las claves E J E M P L O D en un árbol que inicialmente está vacío. La Figura 14.10 muestra el proceso completo del ejemplo cuando se añaden E B U S Q U E D A. El lector debe prestar particular atención a la posición de las claves iguales de este árbol: por ejemplo, aun cuando las E parecen muy alejadas en el árbol, no hay claves «entre» ellas.

La función *ordenar* se obtiene prácticamente de forma gratuita cuando se utiliza un árbol binario de búsqueda, puesto que esta estructura representa a un archivo ordenado, si se mira en el sentido correcto. En las figuras, las claves

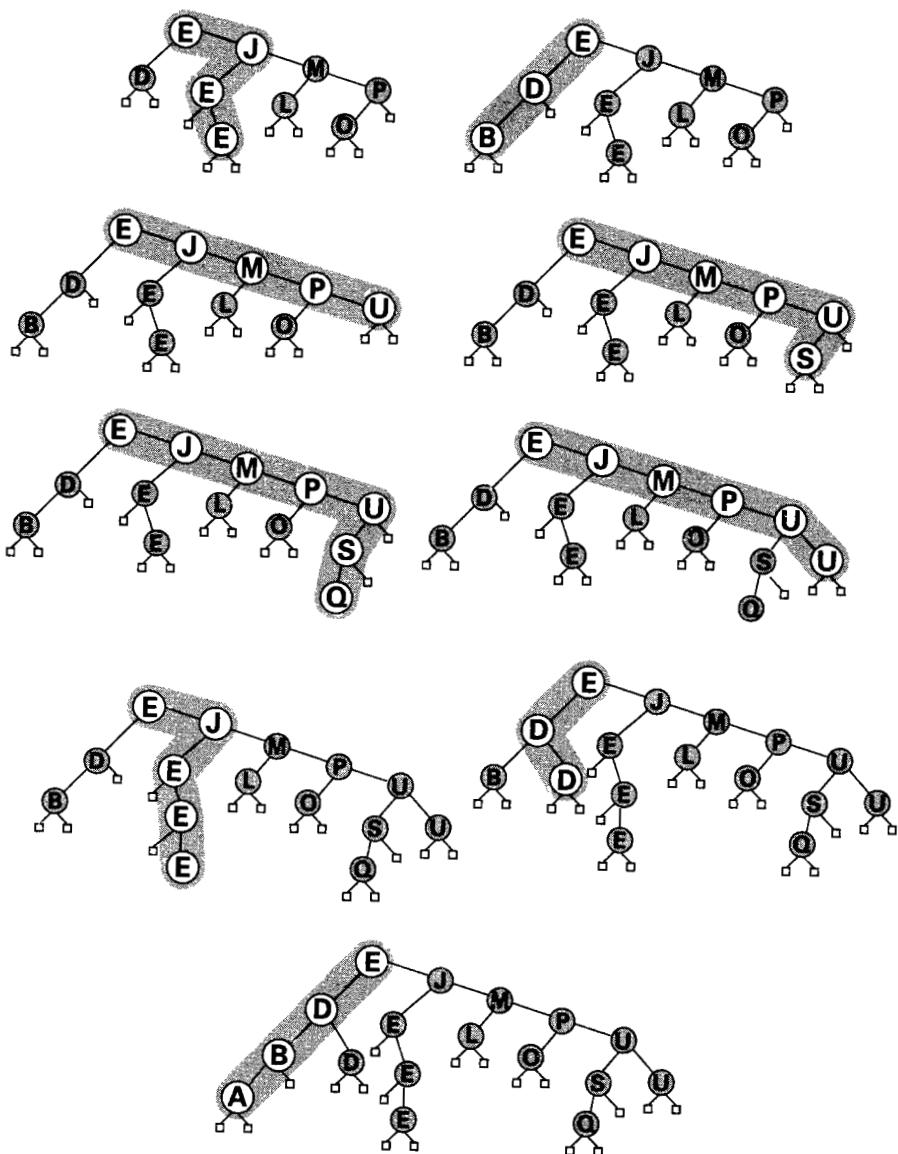


Figura 14.10 Construcción de un árbol binario de búsqueda.

aparecen en orden si se leen de izquierda a derecha de la página (ignorando la altura y los enlaces). Un programa tiene solamente los enlaces para operar, pero el método de ordenación se obtiene directamente de las propiedades que definen a los árboles binarios de búsqueda. Así se define un método de ordenación que es notablemente parecido al Quicksort, con el nodo raíz del árbol desem-

peñando un papel similar al del elemento de partición de la ordenación rápida (no hay claves mayores a la izquierda, no hay claves menores a la derecha). Concretamente, la tarea la llevará a cabo el recorrido recursivo en orden simétrico del Capítulo 5. En este caso se añadiría la operación de clase `Dicc::recorrer()` que simplemente llama a `Dicc::enorden(cabeza->der)` donde `enorden(struct nodo *x)` es la rutina básica del Capítulo 5 (cambiada de nombre) tal que si `x` no es `z`, se llama a sí misma con el argumento `x->izq`, luego llama a `visitar(x)`, luego se llama a sí misma con argumento `x->der`. Así, por ejemplo, si se implementa `Dicc::visitar(struct nodo *x)` para imprimir el campo clave de `x`, una llamada a recorrer imprimiría el árbol entero de forma ordenada. O, como se verá en el Capítulo 27, un `visitar` más intrincado puede llevar a un algoritmo más complicado.

Los tiempos de ejecución de los algoritmos de árboles binarios de búsqueda dependen mucho de la forma de los árboles. En el mejor de los casos, el árbol puede aparecer como el de la Figura 14.3, con aproximadamente $\lg N$ nodos entre la raíz y cada nodo externo. Se puede aspirar a tiempos de búsqueda de orden logarítmico como promedio porque el primer elemento insertado se convierte en la raíz del árbol; si se insertan aleatoriamente N claves, entonces este elemento deberá dividir a las claves en dos partes iguales (de media), y esto produciría tiempos de búsqueda logarítmicos (utilizando el mismo argumento para cada subárbol). En efecto, haciendo abstracción de las claves iguales, podría ocurrir que se produjera un árbol como el dado anteriormente para describir la estructura de comparación de una búsqueda binaria. Éste sería el mejor caso para el algoritmo, que garantizaría tiempos de ejecución logarítmicos para todas las búsquedas. De hecho, en una situación verdaderamente aleatoria, la raíz tiene las mismas posibilidades de ser cualquier clave, así que un árbol perfectamente equilibrado es muy raro. Pero si se insertan claves aleatorias se pueden obtener árboles bastante bien equilibrados.

Propiedad 14.5 Una búsqueda o inserción en un árbol binario de búsqueda requiere alrededor de $2\ln N$ comparaciones, por término medio, en un árbol construido a partir de N claves aleatorias.

Para cada nodo del árbol, el número de comparaciones realizadas para una búsqueda con éxito de dicho nodo es su distancia a la raíz. La suma de estas distancias para todos los nodos se denomina la *longitud del camino interno* del árbol. Dividiendo la longitud del camino interno por N se obtiene el número medio de comparaciones de una búsqueda con éxito. Pero si C_N representa la longitud media del camino interno de un árbol binario de búsqueda de N nodos, se tiene la relación de recurrencia

$$C_N = N - 1 + 1/N \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k})$$

con $C_1 = 1$. (El término $N - 1$ tiene en cuenta el hecho de que la raíz contribuye

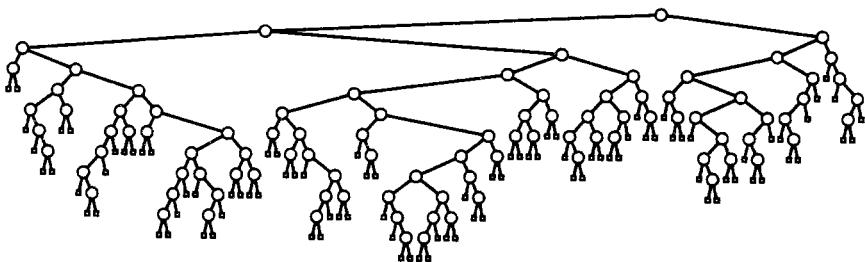


Figura 14.11 Un gran árbol binario de búsqueda.

con 1 a la longitud del camino para cada uno de los restantes $N - 1$ nodos del árbol; el resto de la expresión proviene de observar que la clave de la raíz (la primera insertada) es como si fuera la k -ésima más grande, dejando subárboles aleatorios de tamaño $k-1$ y $N-k$.) Pero esto está muy próximo a la relación de recurrencia que se resolvió en el Capítulo 9 para el Quicksort y se puede resolver de la misma manera para obtener el resultado esperado. El razonamiento para la búsqueda sin éxito es similar, aunque un tanto más complicado. ■

La Figura 14.11 muestra un gran árbol binario de búsqueda construido a partir de una permutación aleatoria de 95 elementos. Aun cuando tiene algunos caminos cortos y algunos largos, se puede decir que está bastante bien equilibrado: cualquier búsqueda necesitará menos de doce comparaciones, y el número «medio» de ellas para encontrar cualquier clave del árbol es 7,00, contra 5,74 de la búsqueda binaria. (El número medio de comparaciones para una búsqueda aleatoria sin éxito es uno más que para la búsqueda con él.) Más aún, se puede insertar una nueva clave por el mismo coste, flexibilidad de la que no se dispone en la búsqueda binaria. Sin embargo, si las claves no están aleatoriamente ordenadas el algoritmo puede tener un mal comportamiento.

Propiedad 14.6 *En el peor caso, una búsqueda en un árbol binario de búsqueda con N claves puede necesitar N comparaciones.*

Por ejemplo, cuando las claves se insertan en orden (o en orden inverso), el método de búsqueda por árbol binario no es mejor que el de búsqueda secuencial descrito al principio de este capítulo. Más aún, hay muchos otros tipos de árboles degenerados que pueden conducir al mismo peor caso (considérese por ejemplo el árbol formado cuando las claves A Z B Y C X ... se insertan en este orden en un árbol inicialmente vacío). En el próximo capítulo se examinará una técnica para eliminar este peor caso y hacer que todos los árboles se parezcan al del caso mejor. ■

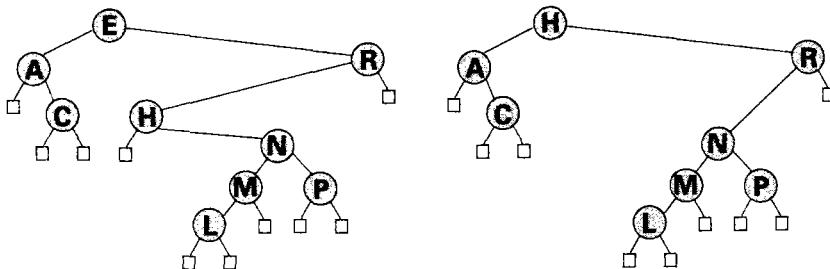


Figura 14.12 Eliminado (de E) de un árbol binario de búsqueda.

Eliminación

Las implementaciones precedentes que utilizan estructuras de árboles binarios para las funciones fundamentales *buscar*, *insertar* y *ordenar* son bastante directas. No obstante, los árboles binarios son un buen ejemplo de un tema recurrente en los algoritmos de búsqueda: la función *eliminar*, que a menudo es bastante incómoda de implementar.

Considérese el árbol que se muestra a la izquierda de la Figura 14.12: eliminar un nodo es fácil si no tiene hijos, como L o P (se «podan» haciendo nulo el correspondiente enlace con su padre); si tiene solamente un hijo, como A, H o R (se desplaza el enlace del hijo al enlace apropiado del padre); o incluso si uno de sus dos hijos no tiene hijos, como N (se utiliza el nodo hijo para reemplazar al padre); pero ¿qué hacer con los nodos más altos del árbol, tales como E?

La Figura 14.12 muestra una forma de eliminar E: se reemplaza por el nodo que tenga la clave superior más próxima (H en este caso). Este nodo forzosamente tiene como mucho un hijo (puesto que no hay nodos entre él y el nodo eliminado, su enlace izquierdo debe ser nulo), y se puede suprimir fácilmente. Así, para quitar E del árbol de la izquierda de la Figura 14.12, se hace apuntar el enlace izquierdo de R al enlace derecho (N) de H, se copian los enlaces del nodo que contiene a E en el que contiene a H, y se hace apuntar `cabeza->der` a H. Esto proporciona el árbol de la derecha de la figura.

El programa que permite tratar todos estos casos es bastante más complejo que las simples rutinas de búsqueda e inserción, pero merece la pena estudiarlo cuidadosamente para preparar las manipulaciones más complejas que se realizarán en el próximo capítulo. El siguiente procedimiento elimina el primer nodo de clave *v* que encuentre en el árbol. (Otra posibilidad es utilizar *info* para identificar el nodo a eliminar.) La variable *p* se utiliza para seguir la pista del parent de *x* en el árbol y la variable *c* se utiliza para encontrar al sucesor del nodo que se va a eliminar. Después de la operación de eliminación, *x* es el hijo de *p*.

```

void Dicc::suprimir(tipoElemento v)
{
    struct nodo *c, *p, *x, *t;
    z->clave = v;
    p = cabeza; x = cabeza->der;
    while (v != x->clave)
        {p = x; x = (v < x->clave) ? x->izq : x->der; }
    t = x;
    if (t->der == z) x = x->izq;
    else if (t->der->izq == z) { x=x->der; x->izq=t->izq; }
    else
    {
        c = x->der; while (c->izq->izq != z) c = c->izq;
        x = c->izq; c->izq = x->der;
        x->izq = t->izq; x->der = t->der;
    }
    delete t;
    if (v < p->clave) p->izq = x; else p->der = x;
}

```

En primer lugar el programa hace una búsqueda en el árbol de forma normal para encontrar el emplazamiento de t en el mismo. (Realmente, el objetivo principal de esta búsqueda es enlazar p con otro nodo una vez que se haya eliminado t .) A continuación el programa verifica tres casos: si t no tiene hijo derecho, entonces el hijo de p después de la eliminación será el hijo izquierdo de t (éste sería el caso de C, L, M, P, y R en la Figura 14.12); si t tiene un hijo derecho que no tiene hijo izquierdo, entonces ese hijo derecho será el hijo de p después de la supresión, con su enlace izquierdo copiado de t (éste sería el caso de A y N en la Figura 14.12); en caso contrario, se pone a x a apuntar al nodo con la clave más pequeña del subárbol de la derecha de t ; el enlace derecho de este nodo se copia en el enlace izquierdo de su padre y sus dos enlaces se ponen a partir de t (éste sería el caso de H y E en la Figura 14.12). Para limitar el número de casos el programa siempre elimina mirando hacia la derecha, aunque en algunos casos podría ser más fácil en algunos casos mirar a la izquierda (por ejemplo, para eliminar a H en la Figura 14.12).

Esta solución puede parecer asimétrica y bastante *ad hoc*: por ejemplo, ¿por qué no utilizar la clave inmediatamente *anterior* a la que se va a eliminar, en lugar de una posterior? Se han sugerido varias modificaciones similares, pero las diferencias no son tan notables como para que puedan apreciarse en aplicaciones prácticas, aunque se ha demostrado que el algoritmo anterior tiende a dejar el árbol ligeramente desequilibrado (con altura media proporcional a \sqrt{N}) si se le somete a un gran número de pares aleatorios de eliminar-insertar.

Es bastante corriente que los algoritmos de búsqueda necesiten implemen-

taciones de la operación de eliminado significativamente complejas: las claves tienden por sí mismas a formar parte integral de la estructura por lo que eliminar una de ellas puede implicar reparaciones complicadas. Una alternativa, a menudo satisfactoria, es la denominada *eliminación perezosa*, en la que el nodo a eliminar se deja en la estructura, pero se marca como «eliminado» para la búsqueda. Esto se obtiene en el programa anterior añadiendo una verificación adicional para detectar tales nodos antes de terminar la búsqueda. Se debe estar seguro de que grandes cantidades de nodos «eliminados» no conducen a un gasto excesivo de tiempo o espacio, pero esto es un problema menor en muchas aplicaciones. De modo alternativo, se puede reconstruir periódicamente toda la estructura de datos, excluyendo los nodos «eliminados»¹.

Árboles binarios de búsqueda indirecta

Como se vio en el Capítulo 11, en muchas aplicaciones se desea obtener una estructura de búsqueda que permita encontrar los registros sin tenerlos que desplazar. Por ejemplo, se puede tener un array de registros con claves y se puede desear que la rutina buscar dé el índice en el array del registro que corresponde con una cierta clave. O se pudiera desear suprimir de la estructura de búsqueda un registro con un índice dado, pero manteniéndolo dentro del array para algún otro uso.

Para adaptar los árboles binarios de búsqueda a tales situaciones, simplemente se transforma el campo `info` de los nodos en el índice del array. Entonces es posible eliminar el campo `clave` haciendo que las rutinas accedan a las claves de los registros directamente, por ejemplo por medio de una instrucción como `if (v < a[x->info])...` Sin embargo, a menudo es mejor hacer una copia extra de las claves y utilizar el código anterior tal cual. Esto implica utilizar una copia suplementaria de las claves (una en el array, otra en el árbol), pero permite que se utilice la misma función para más de un array o, como se verá en el Capítulo 27, para más de un campo clave en el mismo array. (Existen otras vías de lograr esto: por ejemplo, podría asociarse un procedimiento a cada árbol para extraer las claves de los registros.)

Otra forma directa de lograr la «indirección» para los árboles binarios de búsqueda consiste simplemente en suprimir la implementación enlazada y utilizar una representación por array directo, como la que se presentó en el Capítulo 3. Todos los enlaces se convierten en índices dentro de un array `a[0], ..., a[N+1]` de registros que contienen un campo `clave` y campos índices `izq` y `der`. Entonces las referencias a enlaces como `x->clave` y `x = x->izq` pasan a ser referencias a array tales como `a[x].clave` y `x = a[x].izq`. No se utilizan llamadas a `new`, puesto que el árbol existe dentro del array de registros: los no-

¹ De aquí el que el término también signifique *borrado retardado*, porque la acción física de borrar se posterga (se deja para después) o no se hace nunca. (N. del T.)

dos ficticios se asignan colocando `cabeza = 0` y `z = 1` y el constructor incrementa simplemente un puntero al próximo espacio libre dentro del array de registros y rellena los campos.

Esta forma de implementar árboles binarios de búsqueda para facilitar las búsquedas en grandes arrays de registros es preferible en muchas aplicaciones, puesto que evita el gasto extra de copiar las claves descrito en el párrafo anterior y evita la sobrecarga del mecanismo de asignación de memoria que implica `new`. Su inconveniente es que los enlaces no utilizados pueden gastar espacio en el array de registros.

Una tercera alternativa es utilizar arrays paralelos, como se hizo en el Capítulo 3 para las listas enlazadas. La implementación correspondiente es muy parecida a la descrita en el párrafo anterior, excepto que se utilizan tres arrays, uno para las claves, otro para los enlaces izquierdos y otro para los enlaces derechos. La ventaja de este método es la flexibilidad. Se pueden añadir fácilmente nuevos arrays (para la información extra asociada con cada nodo), sin modificar en nada el código de manipulación de los árboles, y cuando una rutina de búsqueda proporciona un índice de un nodo, está dando también una forma inmediata de acceder a todos los arrays.

Ejercicios

1. Implementar un algoritmo de búsqueda secuencial con una media de $N/2$ pasos para una búsqueda cualquiera, con éxito o sin él, manteniendo los registros en un array ordenado.
2. Indicar el orden en que quedan las claves después de insertar los registros con las claves C U E S T I O N F A C I L en una tabla (inicialmente vacía), mediante las operaciones de *buscar* e *insertar* y utilizando una heurística de búsqueda autoorganizada.
3. Dar una implementación recursiva de la búsqueda binaria.
4. Suponiendo que $a[i] == 2*i$, para $1 \leq i \leq N$. ¿Cuántas entradas de la tabla se examinarán por la búsqueda por interpolación durante una búsqueda sin éxito de $2k - 1$?
5. Dibujar el árbol binario de búsqueda que resulta de insertar en un árbol inicialmente vacío los registros de claves C U E S T I O N F A C I L.
6. Escribir un programa recursivo para calcular la *altura* de un árbol binario: la distancia más larga entre la raíz y un nodo externo.
7. Suponiendo que se dispone de una estimación provisional de la frecuencia con la que las claves de búsqueda acceden a un árbol binario. ¿Deberán insertarse las claves en orden creciente o decreciente de dicha frecuencia? ¿Por qué?
8. Modificar un árbol binario de búsqueda de modo que se mantengan juntas en el árbol las claves iguales. (Si varios nodos del árbol tienen la misma clave

que un nodo dado, entonces o su padre o alguno de sus hijos debe tener la misma clave que éste.)

9. Escribir un programa no recursivo para imprimir en orden las claves de un árbol binario de búsqueda.
10. Dibujar el árbol binario de búsqueda que resulte de insertar en un árbol, inicialmente vacío, registros con las claves C U E S T I O N F A C I L, suprimiendo a continuación T.

Árboles equilibrados

Los algoritmos de árboles binarios del capítulo anterior son muy útiles en un gran número de aplicaciones, pero tienen el problema de dar un mal rendimiento en el peor caso. Al igual que con el Quicksort, desgraciadamente es cierto que estos casos tienen tendencia a ocurrir en la práctica si el usuario del algoritmo no se preocupa de ello. El algoritmo de búsqueda de un árbol binario puede comportarse muy mal en archivos ya ordenados, o en archivos en orden inverso, o en archivos que contienen alternativamente claves grandes y pequeñas, o en archivos de gran sección que tengan una estructura simple.

Con el Quicksort el único remedio para mejorar esta situación fue recurrir a la aleatoriedad: escogiendo un elemento que provoque una partición aleatoria, se podría confiar en la ley de las probabilidades para eliminar el peor caso. Afortunadamente, en la búsqueda en árboles binarios es posible hacerlo mucho mejor, pues hay una técnica general que permite *garantizar* que el peor caso no ocurrirá. Esta técnica, a la que se le llama *equilibrar*, ha sido utilizada como base de varios algoritmos diferentes de «árboles equilibrados». A continuación se estudiará con detalle uno de estos algoritmos, presentándose brevemente la forma como se relaciona con los otros métodos que se han utilizado.

Como se verá, la implementación de algoritmos de «árboles equilibrados» es seguramente un caso de «más fácil de decir que de hacer». A menudo es fácil escribir el concepto general que hay detrás del algoritmo, pero la implementación es un conglomerado de casos particulares y simétricos. El programa que se desarrolla en este capítulo no es solamente un método importante de búsqueda sino que ilustra con precisión la relación entre una descripción de «alto nivel» y un programa de «bajo nivel» en C++, que implementa el algoritmo.

Árboles descendentes 2-3-4

Para eliminar el peor caso en los árboles binarios de búsqueda, se necesitará alguna flexibilidad en la estructura de datos que se va a utilizar. Para obtener esta

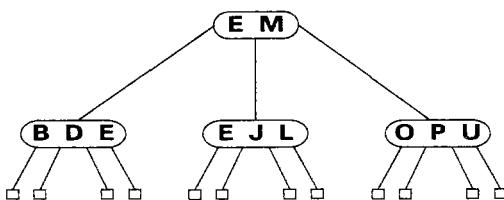


Figura 15.1 Un árbol 2-3-4.

flexibilidad, se supone que los nodos pueden contener más de una clave. Específicamente se permitirán *3-nodos* y *4-nodos*, que pueden contener dos y tres claves, respectivamente. Un 3-nodo tiene tres enlaces saliendo de él, uno para todos los registros con claves más pequeñas que las suyas, otro para todos los registros con claves que están entre las dos suyas, y otro para todos los registros con claves mayores que las suyas. De forma similar un 4-nodo tiene cuatro enlaces que salen de él, uno para cada uno de los intervalos definidos por sus tres claves. (Los nodos de un árbol binario de búsqueda estándar podrían denominarse *2-nodos*: una clave, dos enlaces). Más adelante se verán algunas formas eficaces de definir e implementar las operaciones básicas de estos nodos extendidos; por ahora, supóngase que se pueden manipular convenientemente y observe cómo pueden combinarse para formar árboles.

Por ejemplo, la Figura 15.1 muestra un *árbol 2-3-4* que contiene las claves E J E M P L O D E B U. Es fácil ver cómo se efectúa una búsqueda en este tipo de árbol. Por ejemplo, para buscar S se seguiría el enlace derecho que parte de la raíz, puesto que S es mayor que EM, terminando con una búsqueda sin éxito en el segundo enlace por la derecha del nodo que contiene a O, P y U.

Para insertar un nuevo nodo en un árbol 2-3-4, se desearía, como antes, hacer una búsqueda infructuosa y después enganchar el nodo. Es fácil ver lo que hay que hacer si el nodo en el que termina la búsqueda es un 2-nodo: simplemente, transformarlo en un 3-nodo, añadiéndole el nuevo nodo (y otro enlace). De forma similar, un 3-nodo se puede convertir fácilmente en un 4-nodo. Pero, ¿qué se debe hacer si se desea insertar un nuevo nodo en un 4-nodo? Por ejemplo, ¿cómo se insertaría S en el árbol de la Figura 15.1? Una posibilidad sería engancharlo como un nuevo hijo, el segundo por la derecha del 4-nodo que contiene a O, P y U; pero existe una solución mejor, la que se muestra en la Figura 15.2: se divide el 4-nodo en dos 2-nodos y se pasa una de sus claves a su padre. Primero se divide el 4-nodo que contiene a O, P y U, en dos 2-nodos (uno que contiene a O y el otro a U) y la clave «intermedia» P se pasa hacia arriba, al nodo que contiene a E y M, convirtiéndolo en un 4-nodo. Así hay espacio para S en el 2-nodo que contiene a U.

Pero ¿qué pasa si se divide un 4-nodo cuyo padre es también un 4-nodo? Un método sería dividir también al padre, pero el abuelo también podría ser un 4-nodo y también el bisabuelo, etc.: se tendría que estar haciendo divisiones de nodos hasta la raíz. Una solución más fácil consiste en asegurarse de que el padre de cualquier nodo que se encuentre no sea un 4-nodo, lo que se logra divi-

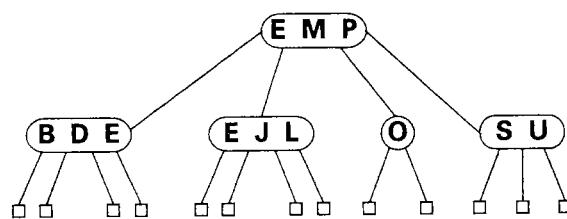


Figura 15.2 Inserción (de S) en un árbol 2-3-4.

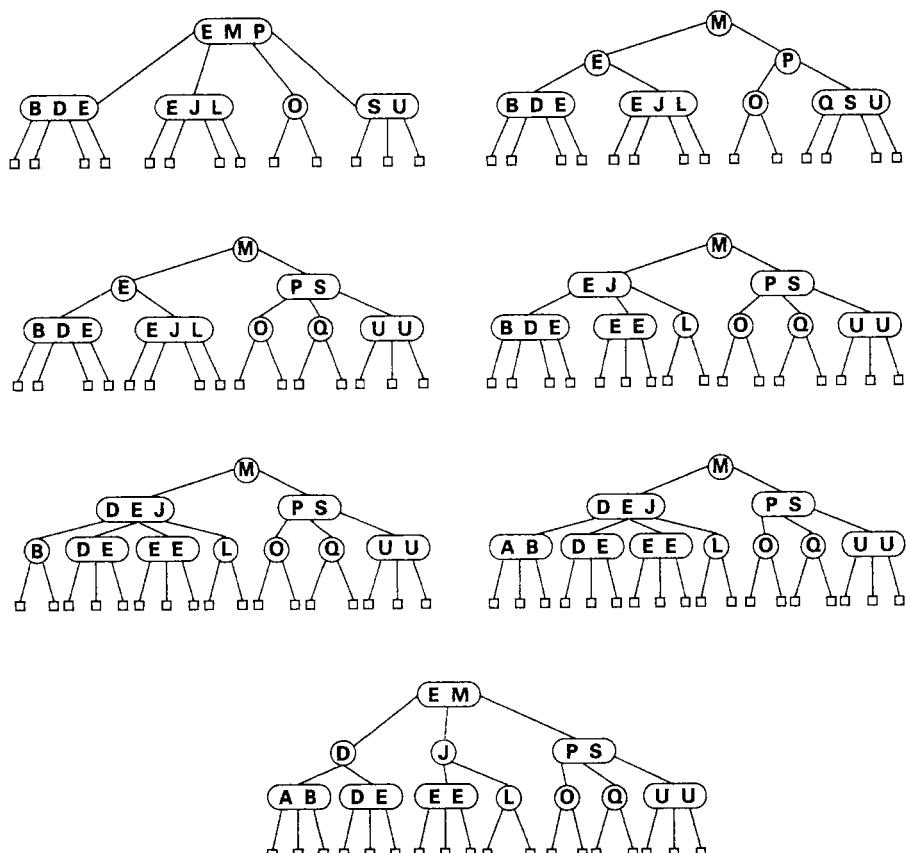


Figura 15.3 Construcción de un árbol 2-3-4.

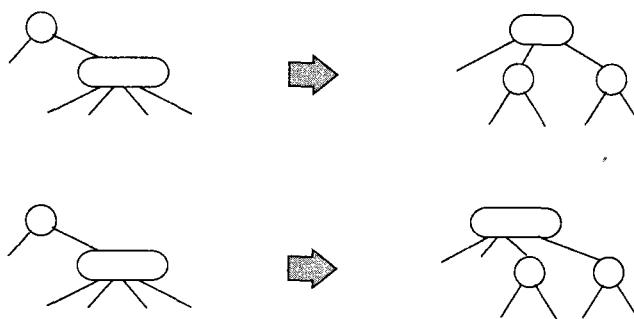


Figura 15.4 División de 4-nodos.

diendo todos los 4-nodos del camino de *descenso* por el árbol. La Figura 15.3 completa la construcción de un árbol 2-3-4 correspondiente al conjunto completo de claves E J E M P L O D E B U S Q U E D A. En la primera línea se ve que el nodo raíz se divide durante la inserción de la Q; se producen otras divisiones al insertar la segunda U, la última E y la segunda D.

El ejemplo anterior muestra cómo se pueden insertar fácilmente nuevos nodos en árboles 2-3-4 haciendo una búsqueda y dividiendo los 4-nodos que se encuentran en el descenso por el árbol. De forma más precisa, como se muestra en la Figura 15.4, cada vez que se encuentre un 2-nodo conectado con un 4-nodo, se debe transformar en un 3-nodo conectado con dos 2-nodos, y cada vez que se encuentre un 3-nodo conectado con un 4-nodo, se debe transformar en un 4-nodo conectado a dos 2-nodos.

Esta operación de «división» funciona porque se pueden mover no sólo las claves sino también los *punteros*. Dos 2-nodos tienen el mismo número de punteros (cuatro) que un 4-nodo, así que se puede efectuar la división sin tener que transformar los elementos que están debajo del nodo dividido. Un 3-nodo no puede transformarse en un 4-nodo añadiendo solamente otra clave: se necesita también otro puntero (en este caso el puntero extra liberado por la división). El punto crucial es que estas transformaciones son puramente «locales»: las únicas partes del árbol que se necesita examinar o modificar son las que se muestran en la Figura 15.4. Cada una de las transformaciones transmite hacia arriba una de las claves, desde un 4-nodo hacia su padre y reestructura los enlaces de acuerdo con ello.

Hay que destacar que no es necesario preocuparse por saber si el padre de un nodo es un 4-nodo, puesto que las transformaciones aseguran que cuando se pasa a través de un nodo durante el descenso del árbol, se acaba desembocando en un nodo que no es un 4-nodo. En particular, cuando se alcanza el fondo del árbol no se está en un 4-nodo, y se puede insertar un nuevo nodo directamente transformando un 2-nodo en un 3-nodo o un 3-nodo en un 4-nodo. En realidad, es conveniente tratar la inserción como una división de un

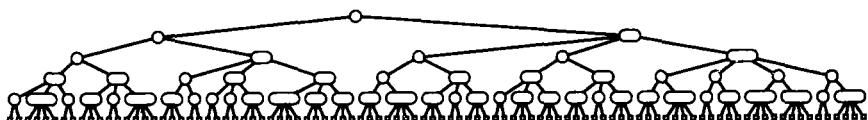


Figura 15.5 Un árbol 2-3-4 grande.

4-nodo imaginario del fondo del árbol, el cual transmite hacia arriba la nueva clave a insertar.

Un último detalle: siempre que la raíz del árbol pase a ser un 4-nodo, se le transforma en tres 2-nodos, como se hizo en el ejemplo anterior. Esta técnica es más simple que la alternativa de estar esperando hasta la próxima inserción para hacer la división, porque no es necesario preocuparse por el padre de la raíz. La división de la raíz (y sólo esta operación) hace que el árbol crezca un nivel «más alto».

El algoritmo esquematizado anteriormente proporciona un camino para hacer búsquedas e inserciones en árboles 2-3-4; puesto que los 4-nodos se dividen en el camino de descenso, los árboles se denominan *árboles 2-3-4 descendentes*. Lo interesante es que, aun cuando no ha habido que preocuparse por el equilibrio, los árboles resultantes ¡están perfectamente equilibrados!

Propiedad 15.1 *Las búsquedas en un árbol 2-3-4 de N nodos nunca exploran más de $\lg N + 1$ nodos.*

La distancia de la raíz a cualquier nodo externo es la misma: las transformaciones que se llevan a cabo no tienen influencia sobre la distancia entre cualquier nodo y la raíz, excepto cuando se divide la raíz y, en este caso, la distancia entre todos los nodos y la raíz se aumenta en una unidad. Si todos los nodos son 2-nodos, el resultado anunciado se cumple puesto que el árbol es similar a un árbol binario completo; si existen 3-nodos o 4-nodos, entonces la altura del árbol sólo puede ser inferior. ■

Propiedad 15.2 *Las inserciones en árboles 2-3-4 de N nodos necesitan menos de $\lg N + 1$ divisiones de nodos en el peor caso y parecen necesitar menos de una división por término medio.*

Lo peor que puede pasar es que todos los nodos en el camino hacia el punto de inserción sean 4-nodos, que sería preciso dividir. Pero en un árbol construido a partir de permutaciones aleatorias de N elementos, no sólo es improbable que suceda el peor caso, sino que también se necesitan pocas divisiones por término medio, porque no hay muchos 4-nodos. La Figura 15.5 muestra un árbol construido a partir de una permutación aleatoria de 95 elementos: hay nueve 4-nodos y sólo uno de éstos no está situado en el fondo. Los expertos no han podido

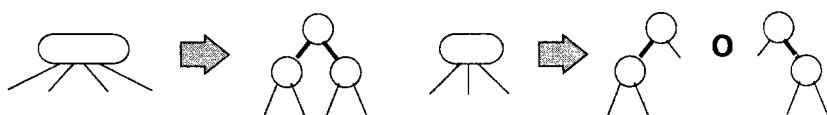


Figura 15.6 Representación rojinegra de 3-nodos y 4-nodos.

establecer todavía resultados analíticos del rendimiento medio de los árboles 2-3-4, pero los estudios empíricos muestran de forma insistente que se necesitan pocas divisiones.■

La descripción precedente es suficiente para definir un algoritmo de búsqueda utilizando árboles 2-3-4 que garanticen un buen rendimiento en el peor caso. Sin embargo, sólo se está a mitad de camino de una implementación real. Aunque sea posible escribir algoritmos que realmente lleven a cabo transformaciones sobre distintos tipos de datos destinados a representar 2-, 3-, y 4-nodos, la mayor parte de las acciones a tomar son poco prácticas en esta representación directa. (Para convencerte de esto es suficiente con tratar de implementar incluso la más simple de las transformaciones de dos nodos.) Además, el gasto extra en que se incurre por la manipulación de estructuras de nodos más complejas es muy probable que haga a los algoritmos más lentos que la búsqueda estándar por árbol binario. El objetivo principal la acción de equilibrar es ofrecer un «seguro» contra el peor caso, pero sería penoso tener que pagar el coste adicional de este seguro en cada ejecución del algoritmo. Por fortuna, como se verá más adelante, existe una representación relativamente simple de los 2-, 3- y 4-nodos que permite que las transformaciones se hagan de manera uniforme con un pequeño aumento de coste respecto al de una búsqueda estándar por árbol binario.

Árboles rojinegros

Curiosamente, es posible representar los árboles 2-3-4 como árboles binarios estándar (2-nodos solamente) utilizando sólo un bit extra por nodo. La idea es representar los 3-nodos y los 4-nodos como pequeños árboles binarios unidos por enlaces «rojos» que contrasten con los enlaces «negros» que ligan a los árboles 2-3-4. La representación es simple: como se muestra en la Figura 15.6, los 4-nodos se representan por 2-nodos conectados por un enlace rojo (los enlaces rojos se dibujarán con líneas gruesas). (Cualquier orientación es legal para un 3-nodo.)

La Figura 15.7 muestra una forma de representar el último árbol de la Fi-

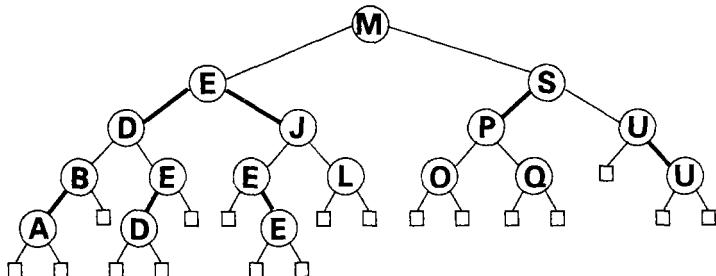


Figura 15.7 Un árbol rojinegro.

gura 15.3. Si se eliminan los enlaces rojos y se reúnen los nodos que conectan, el resultado será el árbol 2-3-4 de la Figura 15.3. El bit extra por nodo se utiliza para almacenar el color del enlace que apunta a ese nodo: se hará referencia a los árboles 2-3-4 representados de esta forma con el nombre de *árboles rojinegros*.

La «inclinación» de cada 3-nodo se determina por la dinámica del algoritmo que se describe posteriormente. Existen muchos árboles rojinegros para cada árbol 2-3-4. Sería posible hacer cumplir una regla para que los 3-nodos tengan todos la misma inclinación, pero no hay razón para hacerlo así.

Estos árboles tienen muchas propiedades que se obtienen directamente de la forma en la que se han definido. Por ejemplo, nunca hay dos enlaces rojos seguidos a lo largo de cualquier camino entre la raíz y un nodo terminal, y todos los caminos de este tipo tienen el mismo número de enlaces negros. Es de destacar que es posible que un camino (alternando negro-rojo) puede ser dos veces más largo que otro (todo negro), pero las longitudes de los caminos son siempre proporcionales a $\log N$.

Una característica sorprendente de la Figura 15.7 es la posición relativa de las claves iguales. Con un poco de reflexión quedará claro que cualquier algoritmo de árbol equilibrado debe permitir que los registros con claves iguales a la de un nodo dado se encuentren a ambos lados del nodo: de lo contrario se podría producir un grave desequilibrio que provocaría la inserción de largas cadenas de claves duplicadas. Esto implica que no es posible encontrar todos los nodos que tengan una clave dada continuando con el procedimiento de búsqueda, como en la búsqueda estándar por árboles binarios. En su lugar, se debe utilizar un procedimiento como el de imprimir un árbol del Capítulo 14, o bien eliminar la posibilidad de que haya claves duplicadas, como se presentó al comienzo del Capítulo 14.

Una propiedad muy agradable de los árboles rojinegros es que el procedimiento buscar para la búsqueda estándar por árboles binarios se aplica en ellos sin modificaciones (excepto en el caso de las claves duplicadas que se presentó en el párrafo anterior). Se implementan los colores de los enlaces añadiendo un campo *b* de un bit a cada nodo. Este campo será 1 si el enlace que apunta al

nodo es rojo y 0 si es negro; el procedimiento `buscar` nunca examina este campo. De esta manera, el mecanismo de equilibrado no añade ninguna «sobrecarga» al tiempo empleado por el procedimiento fundamental de búsqueda. Puesto que en una aplicación típica cada clave se inserta una sola vez, pero puede buscarse muchas veces, el resultado final será que se ha mejorado el tiempo de búsqueda (porque los árboles están equilibrados) a un coste relativamente pequeño (porque no se ha hecho ninguna acción de equilibrar durante las búsquedas).

Más aún, el sobrecoste de la inserción es muy pequeño: solamente hay que hacer algo diferente al alcanzar un 4-nodo, y no hay muchos 4-nodos en el árbol porque siempre se están dividiendo. El lazo interno necesita sólo una comprobación extra (si un nodo tiene dos hijos rojos, es parte de un 4-nodo), tal y como se muestra en la siguiente implementación del procedimiento `insertar`:

```
void Dicc::insertar(tipoElemento v, tipoInfo info)
{
    x = cabeza; p = cabeza; a = cabeza;
    while (x != z)
    {
        ba = a; a = p; p = x;
        x = (v < x->clave) ? x->izq : x->der;
        if (x->izq->b && x->der->b) dividir(v);
    }
    x = new nodo(v, info, 1, z, z);
    if (v < p->clave) p->izq = x; else p->der = x;
    dividir(v); cabeza->der->b = negro;
}
```

Por claridad, se utilizan las constantes `rojo` = 1 y `negro` = 0, en éste y en los siguientes códigos; por brevedad se comprueba el 1 comprobando un no cero, sin hacer referencia al rojo. En este programa `x` se mueve hacia abajo por el árbol al igual que antes, y `ba`, `a` y `p` se mantienen como punteros al bisabuelo, al abuelo y al padre de `x` en el árbol. Para comprender por qué se necesitan todos estos enlaces, se considera la adición de `Y` al árbol de la Figura 15.7. Cuando se alcanza el nodo externo de la derecha del 3-nodo que contiene a `UU`, `ba` es `S`, `a` es `U` y `p` es `U`. Ahora se debe añadir `Y` para hacer un 4-nodo que contenga `U`, `U` y `Y`, resultando el árbol que se muestra en la Figura 15.8.

Se necesita un puntero a `S` (`ba`) porque su enlace derecho debe cambiar para que apunte a la segunda `U` y no a la primera. Para ver exactamente cómo sucede esto, se necesita examinar la operación del procedimiento `dividir`. Considerese la representación rojinegra de las dos transformaciones que se deben llevar a cabo: si se tiene un 2-nodo conectado con un 4-nodo, entonces hay que convertirlos en un 3-nodo conectado con dos 2-nodos; si se tiene un 3-nodo conectado a un 4-nodo hay que convertirlos en un 4-nodo conectado a dos 2-nodos.

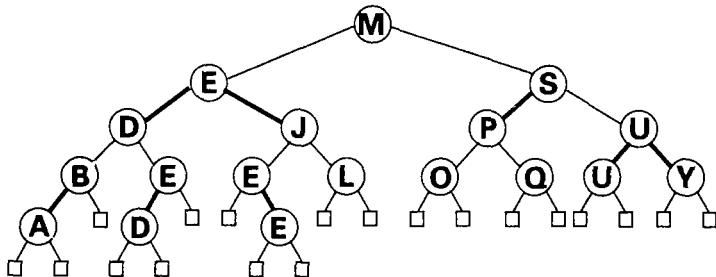


Figura 15.8 Inserción (de Y) en un árbol rojinegro.

dos. Cuando se añade un nuevo nodo en la parte inferior del árbol, se le considera como el nodo intermedio de un 4-nodo imaginario (esto es, imaginando que z es rojo, aunque por ello nunca se le compruebe explícitamente).

La transformación necesaria cuando se encuentra un 2-nodo conectado a un 4-nodo es fácil y se aplica también si se tiene un 3-nodo conectado a un 4-nodo de forma «correcta», como se muestra en la Figura 15.9. Así, dividir comienza marcando a x como rojo y a sus hijos como negros.

No queda más que considerar las otras dos situaciones que pueden ocurrir si se encuentra un 3-nodo conectado a un 4-nodo, como se muestra en la Figura 15.10 (realmente, hay cuatro situaciones, dado que se pueden dar también las imágenes simétricas de estas dos en la otra orientación de los 3-nodos). En estos casos la división del 4-nodo deja dos enlaces rojos seguidos, una situación ilegal que debe corregirse. Esto se puede comprobar fácilmente dentro del propio programa: como se acaba de marcar a x como rojo, sólo se deberá actuar si el padre p de x es también rojo. La situación no es tan grave porque se tienen tres nodos conectados por enlaces rojos: todo lo que hay que hacer es transformar el árbol de modo que los enlaces rojos apunten hacia abajo desde el propio nodo.

Por fortuna, existe una operación simple que logra el efecto deseado. Se co-

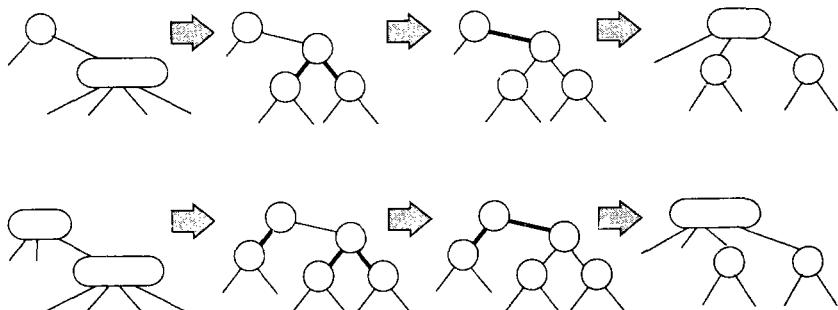


Figura 15.9 División de 4-nodos con un cambio de colores.

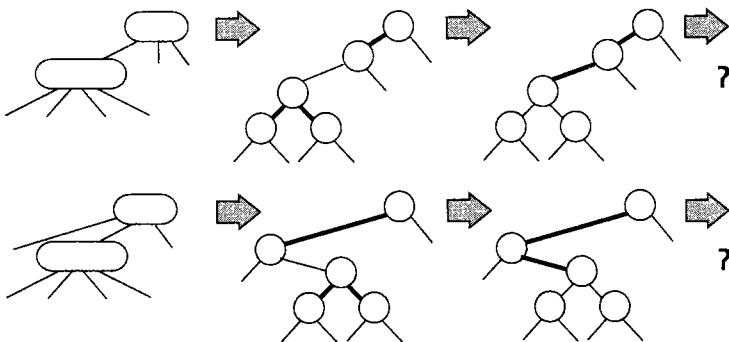


Figura 15.10 División de 4-nodos con un cambio de colores: se necesita una rotación.

mienza por la más fácil de las dos situaciones, el primer caso (parte superior) de la Figura 15.10, en el que los enlaces rojos están orientados en la misma dirección. El problema es que el 3-nodo se orientó en la dirección equivocada: en consecuencia se reestructurará el árbol para cambiar la orientación del 3-nodo y así reducir este caso al segundo de la Figura 15.9, en el que la marca de color de x y sus hijos fue suficiente. Al reestructurar el árbol para reorientar un 3-nodo se cambian tres enlaces, como se muestra en la Figura 15.11; en esta figura el árbol de la izquierda es el mismo que el que se obtuvo en la Figura 15.8, pero en el de la derecha está girado el 3-nodo que contiene a P y a S. El enlace izquierdo de S se cambió para apuntar a Q, el enlace derecho de P se cambió para apuntar a S y el enlace derecho de M se cambió para apuntar a P. Se observa que los colores de los dos nodos también se cambiaron.

Esta operación de *rotación simple* se define sobre cualquier árbol binario de búsqueda (con la excepción de las operaciones que afectan a colores) y es la base de varios algoritmos de árboles equilibrados, porque preserva el carácter esencial del árbol de búsqueda y es una modificación local que implica sólo tres cambios de enlaces. Sin embargo, es importante observar que la aplicación de una rotación simple no mejora necesariamente el equilibrio de un árbol. En la

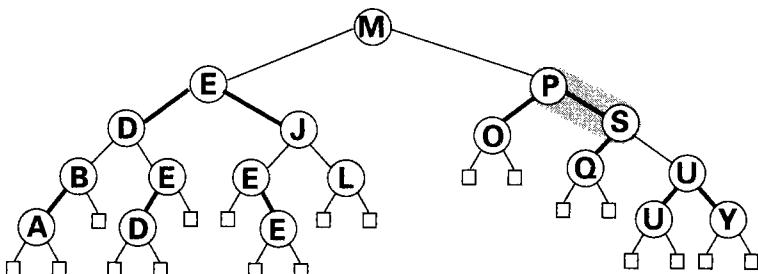


Figura 15.11 Rotación de un 3-nodo de la Figura 15.8.

Figura 15.11, la rotación hace subir un nivel hacia la raíz a todos los nodos a la izquierda de P, pero todos los nodos a la derecha de S han *bajado* un nivel: en este caso la rotación hace al árbol menos, no más, equilibrado. Los árboles 2-3-4 descendentes se pueden considerar simplemente como una forma conveniente de identificar rotaciones simples que *son* susceptibles de mejorar el equilibrio.

Toda rotación simple implica modificar la estructura del árbol, algo que se debe hacer con precaución. Como se vio al considerar el algoritmo de eliminación del Capítulo 14, el código es más complicado de lo que pudiera parecer necesario a causa del gran número de casos similares con simetrías izquierdadera. Por ejemplo, suponiendo que los enlaces y, h, y n apuntan a M, S y P respectivamente, en la Figura 15.8, entonces la transformación para pasar a la Figura 15.11 se efectúa por los cambios de enlace $h \rightarrow izq = n \rightarrow der; n \rightarrow der = h; y \rightarrow der = n$. Existen otros tres casos análogos: el 3-nodo podría estar orientado en el otro sentido, o podría estar en el lado izquierdo de y (orientado en ambos sentidos). Una forma práctica de tratar estos cuatro casos es utilizar la clave de búsqueda v para «redescubrir» el hijo (h) y el nieto (n) del nodo y. (Se sabe que solamente se reorientará un 3-nodo si la búsqueda llevó al último nivel del árbol.) Esto conduce a un programa más simple que la alternativa de estar recordando durante la búsqueda no sólo los dos enlaces correspondientes a h y n, sino también si son derechos o izquierdos. La siguiente función permite reorientar un 3-nodo a lo largo del camino de búsqueda de v, cuyo padre es y:

```
struct nodo *rotar(tipoElemento v, struct nodo *y)
{
    struct nodo *h, *n;
    h = (v < y->clave) ? y->izq : y->der;
    if (v < h->clave)
        { n = h->izq; h->izq = n->der; n->der = h; }
    else
        { n = h->der; h->der = n->izq; n->izq = h; }
    if (v < y->clave) y->izq = n; else y->der = n;
    return n;
}
```

Si y apunta a la raíz, h es el enlace derecho de y y n es el enlace izquierdo de h, el programa realiza exactamente las trasformaciones de enlaces necesarias para producir el árbol de la Figura 15.11 a partir del de la Figura 15.8. El lector puede verificar por sí mismo los otros casos. Esta función devuelve el enlace del «tope» del 3-nodo, pero no hace el cambio de color.

Así, para tratar el tercer caso de dividir (ver Figura 15.10), se puede hacer a rojo, luego asignar a x rotar(v, ba), y después poner x en negro. Esto reorienta el 3-nodo constituido por los dos nodos a los que apuntan a y p y hace que este caso sea el mismo que el segundo, cuando se orientó el 3-nodo.

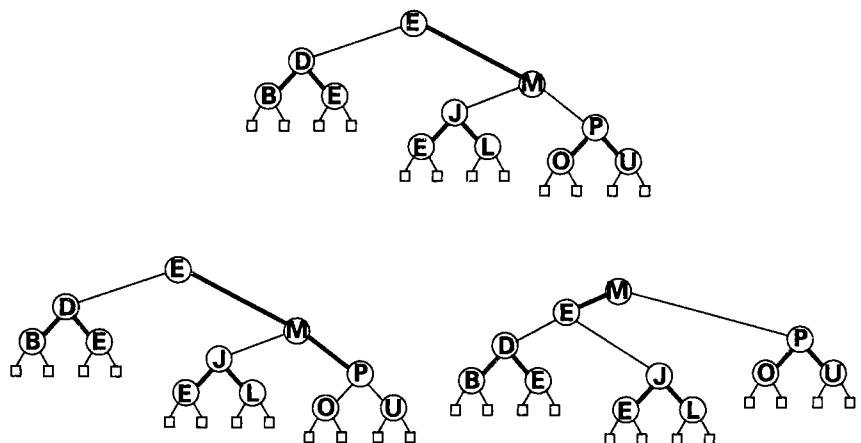


Figura 15.12 División de un nodo en un árbol rojinegro.

Por último, para tratar el caso en el que los dos enlaces rojos están orientados en direcciones diferentes (ver Figura 15.10), simplemente se pone en p el valor de $\text{rotar}(v, a)$. Esto reorienta el 3-nodo «ilegal» constituido por los dos nodos a los que apuntan p y x . Estos nodos son del mismo color, así que no es necesario ningún cambio de color, lo que lleva directamente al tercer caso. Por razones evidentes, la combinación de lo que se acaba de presentar y de la rotación relativa al tercer caso se denomina *rotación doble*.

La Figura 15.12 muestra la acción de dividir en el ejemplo cuando se añade S . En primer lugar existe un cambio de color para dividir el 4-nodo que contiene a O , P y U . A continuación es preciso hacer una rotación doble: la primera alrededor de la arista entre P y M y la segunda alrededor de la arista entre M y E . Como los dos enlaces rojos se hallan en la misma dirección se está en el tercer caso. Después de las modificaciones, se puede insertar S a la izquierda de U , como se muestra en el primer árbol de la Figura 15.13. Si la raíz es un 4-nodo (inserción en el primer árbol de la Figura 15.13) entonces el procedimiento *dividir* pone la raíz en rojo: esto corresponde a transformarla, junto con el nodo ficticio que está encima de ella, en un 3-nodo. Evidentemente, no hay razón para hacer esto, por lo que se incluye una sentencia al final del código de inserción, que permite mantener a la raíz en negro.

Esto completa la descripción de las operaciones que debe llevar a cabo *dividir*. Este procedimiento debe cambiar el color de x y de su hijo, efectuar la parte inferior de una rotación doble, si es necesario, y a continuación efectuar una rotación simple, también si es necesario, de la siguiente forma:

```
void dividir(tipoElemento v)
{
    x->b = rojo; x->izq->b = negro; x->der->b = negro;
```

```

if (p->b)
{
    a->b = rojo;
    if (v < a->clave != v < p->clave) p = rotar(v,a);
    x = rotar(v, ba);
    x->b = negro;
}
}

```

Este procedimiento fija los colores después de la rotación y también reinicializa a x lo suficientemente alto en el árbol para asegurar que la búsqueda no se pierda debido a todos los cambios de los enlaces.

Los códigos de dividir y rotar se han incluido en procedimientos separados por razones de claridad, utilizando las variables ba, etc., de insertar; en C++ son posibles diversas alternativas menos atractivas, que van desde hacerlas globales hasta declararlas explícitamente como funciones friends de Dicc.

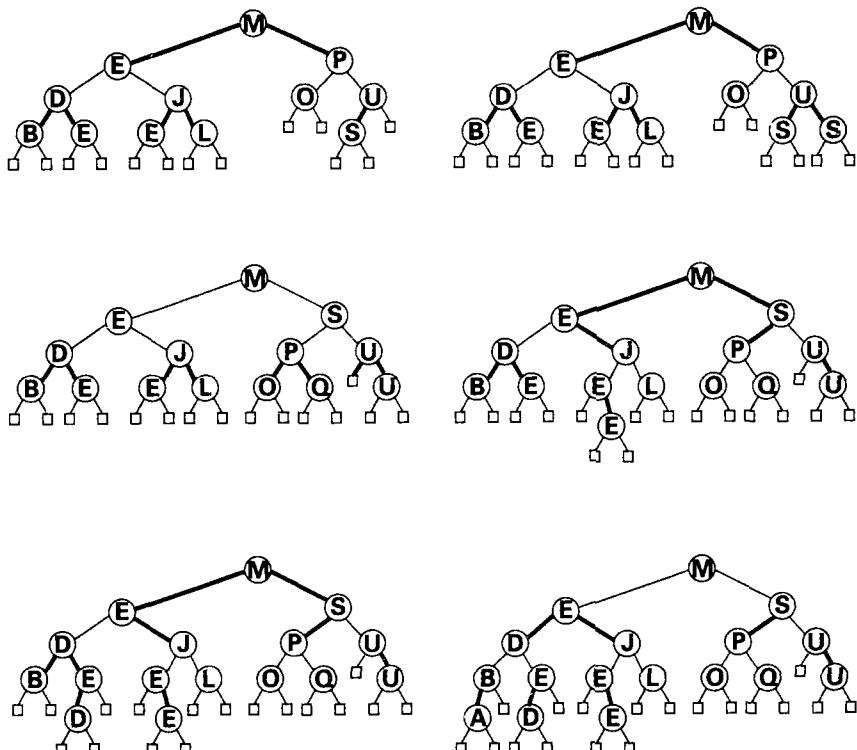


Figura 15.13 Construcción de un árbol rojinegro.

La declaración de clase para los árboles rojinegros es la misma que la que se dio en el capítulo anterior para los árboles binarios de búsqueda normales, con la adición del campo indicador binario `b` en `nodo` y la inclusión de un argumento en el constructor de `nodo` para su inicialización. Si hay espacio disponible, se podría declarar `b` como un entero, pero normalmente se intenta utilizar solamente un bit, quizás el de signo de una clave entera o el de alguna parte del registro que se ha denominado `info`. A continuación se deben inicializar cuidadosamente los nodos ficticios del constructor de `Dicc`, como sigue:

```
Dicc(int max)
{
    z = new nodo(0, infoNIL, negro, 0, 0);
    z->izq = z; z->der = z;
    cabeza = new nodo(elementoMIN, 0, negro, 0, z);
}
```

Los enlaces de `z` se ponen apuntando a `z`. Aunque no es común que se necesiten obligatoriamente asignaciones tales como un centinela, pueden simplificar bastante la codificación. Por ejemplo, estas asignaciones permiten evitar el uso de un `break` en el lazo `while` de `insertar`.

Al ensamblar los distintos fragmentos de código anteriores se obtiene un algoritmo muy eficaz y relativamente simple para la inserción utilizando una estructura de árbol binario que garantiza toda inserción o búsqueda en un número de pasos logarítmico. Éste es uno de los pocos algoritmos de búsqueda con esta propiedad, y su utilización está justificada siempre que no se pueda tolerar un mal rendimiento en el peor caso.

La Figura 15.13 muestra cómo este algoritmo construye el resto del árbol rojinegro del conjunto de claves del ejemplo. Por el coste de solamente unas pocas rotaciones se obtiene un árbol bastante bien equilibrado.

Propiedad 15.3 *Una búsqueda en un árbol rojinegro de N nodos construido a partir de claves aleatorias parece necesitar $\lg N$ comparaciones y una inserción parece necesitar, como media, menos de una rotación.*

A pesar de que todavía está por hacer un análisis preciso del caso medio de este algoritmo, existen resultados convincentes de análisis parciales y de simulaciones. La Figura 15.14 muestra el gran árbol construido para el ejemplo que se ha venido utilizando: el número medio de nodos explorados en este árbol durante la búsqueda de una clave aleatoria es apenas 5,81, en comparación con 7,00 del árbol construido para las mismas claves en el Capítulo 14, y con 5,74, el valor óptimo para un árbol perfectamente equilibrado.■

Pero el significado real de los árboles rojinegros se encuentra en su rendimiento en el peor caso y en el hecho de que este rendimiento se alcanza con muy poco coste. La Figura 15.15 muestra el árbol construido si se insertan los

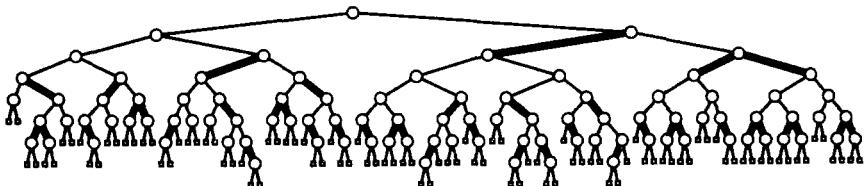


Figura 15.14 Un gran árbol rojinegro.

números del 1 al 95, en orden, en un árbol inicialmente vacío; incluso este árbol está bastante bien equilibrado. El coste de la búsqueda para cada nodo del árbol es tan bajo como si se hubiera construido por el algoritmo elemental y la inserción solamente implica un bit extra de comprobación y alguna llamada ocasional al procedimiento dividir.

Propiedad 15.4 *Una búsqueda en un árbol rojinegro con N nodos necesita menos de $2\lg N + 2$ comparaciones y una inserción necesita menos de una cuarta parte de rotaciones que de comparaciones.*

Sólo las «divisiones» que corresponden a un 3-nodo conectado a un 4-nodo en un árbol 2-3-4 necesitan una rotación en el árbol rojinegro correspondiente, por lo que esta propiedad se deduce de la propiedad 15.2. El peor caso se obtiene cuando el camino al punto de inserción consiste en alternar 3- y 4-nodos.■

En resumen: utilizando este método se puede encontrar una clave de un archivo de, por ejemplo, medio millón de registros al compararlos con sólo otras veinte claves. En el peor caso puede ser que sean necesarias dos veces más comparaciones, pero no más. Además, las comparaciones se acompañan de un sobrecoste pequeño, lo que asegura una búsqueda muy rápida.

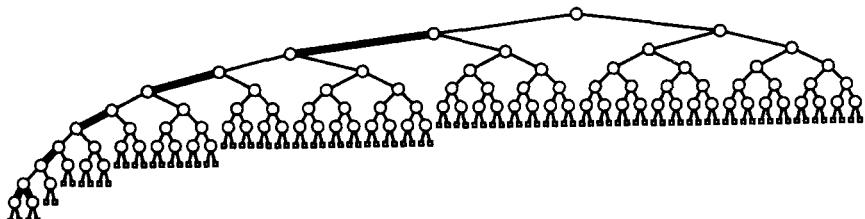


Figura 15.15 Un árbol rojinegro en un caso degenerado.

Otros algoritmos

La implementación del «árbol 2-3-4 descendente», utilizando el esquema rojinegro presentado en la sección anterior, es una de las diversas estrategias similares que se han propuesto para implementar árboles binarios equilibrados. Como se vio anteriormente, de hecho es la operación «rotar» la que equilibra los árboles: se han examinado los árboles desde un enfoque particular que permite decidir fácilmente cuándo efectuar la rotación. Otros enfoques de los árboles conducen a otros algoritmos, algunos de los cuales se examinarán brevemente.

La estructura de datos más antigua y la mejor conocida para los árboles equilibrados es el *árbol AVL*. Estos árboles tienen la propiedad de que la altura de los dos subárboles de cada nodo difieren a lo sumo en una unidad. Si esta condición se viola por una inserción, se puede restaurar utilizando rotaciones. Pero esto requiere un bucle extra: el algoritmo básico consiste en buscar el valor a insertar y después seguir *hacia arriba* por el árbol, a lo largo del camino que se está recorriendo, ajustando las alturas de los nodos utilizando rotaciones. También es necesario saber si cada nodo tiene una altura superior o inferior en una unidad a la de su hermano, o es la misma. Esto necesita dos bits si se recurre a una implementación directa, aunque exista una forma de lograrlo con un solo bit por nodo, utilizando el modelo rojinegro.

Una segunda, y muy conocida, estructura de árbol equilibrado es el árbol 2-3, en el que sólo se permiten 2-nodos y 3-nodos. Es posible implementar *inserción* utilizando un «bucle extra» que incluya rotaciones, como para los árboles AVL, pero estas estructuras no tienen la suficiente flexibilidad para que se pueda dar una versión descendente válida del algoritmo. Una vez más, el modelo rojinegro puede simplificar la implementación, pero es preferible utilizar *árboles 2-3-4 ascendentes*, en los que se busca descendiendo hasta el fondo del árbol para hacer allí la inserción, y entonces (si el nodo del fondo es un 4-nodo) se vuelve hacia atrás ascendiendo por el camino de búsqueda, dividiendo los 4-nodos e insertando el nodo intermedio en el padre, hasta encontrar un 2-nodo o un 3-nodo como parente. En este punto puede ser necesaria una rotación para tratar casos como los de la Figura 15.10. Este método tiene la ventaja de utilizar a lo sumo una rotación por inserción, lo que puede ser muy importante en algunas aplicaciones. La implementación es algo más complicada que la del método descendente descrito anteriormente.

En el Capítulo 18, se estudiará el tipo más importante de árbol equilibrado, una generalización de los árboles 2-3-4 denominada *árboles B*. Esta estructura permite hasta M claves por nodo, siendo M grande. Estos árboles se utilizan ampliamente en aplicaciones que trabajan con archivos muy grandes.

Ejercicios

1. Dibujar el árbol 2-3-4 descendente construido por inserción de las claves C U E S T I O N F A C I L (en este orden) en un árbol inicialmente vacío.
2. Dibujar una representación rojinegra del árbol de la pregunta anterior.
3. ¿Qué enlaces se modifican exactamente por los procedimientos dividir y rotar cuando se inserta Z (después de Y) en el árbol ejemplo de este capítulo?
4. Dibujar el árbol rojinegro que resulte de la inserción de las letras A hasta la K (en este orden), describiendo lo que pasa en general cuando las claves se insertan en los árboles en orden ascendente.
5. ¿Cuántos enlaces deben modificarse en una rotación doble y cuántos se modificaron en la implementación dada?
6. Generar aleatoriamente dos árboles rojinegros de 32 nodos, dibujándolos (a mano o por un programa), y compararlos con los árboles binarios de búsqueda no equilibrados construidos con las mismas claves.
7. Generar aleatoriamente diez árboles rojinegros de 1.000 nodos. Calcular el número de rotaciones necesarias para construir los árboles y la longitud media del camino entre la raíz y un nodo externo. Interpretar los resultados.
8. Con un bit por nodo para el «color» se pueden representar 2-nodos, 3-nodos y 4-nodos. ¿Cuántos tipos de nodos diferentes se podrían representar si se utilizan dos bits por nodo para el color?
9. En los árboles rojinegros se necesitan las rotaciones cuando los 3-nodos se convierten en 4-nodos de una forma «no equilibrada». ¿Por qué no se eliminan las rotaciones permitiendo que los 4-nodos se representen como tres nodos cualquiera conectados por dos enlaces rojos (perfectamente equilibrados o no)?
10. Determinar una secuencia de inserciones que construya el árbol rojinegro que se muestra en la Figura 15.11.

Dispersión

Una técnica de búsqueda completamente diferente de las basadas en estructuras de árboles de comparación de los capítulos anteriores es la *dispersión*: un método que permite hacer directamente referencia a los registros de una tabla por medio de transformaciones aritméticas sobre las claves para obtener direcciones de la tabla. Si se sabe que las claves son enteros distintos, entre 1 y N , entonces se puede almacenar un registro con clave i en la posición i de la tabla, preparado para que se acceda a él de forma inmediata con el valor de la clave. La *dispersión* es una generalización de este método trivial en aplicaciones de búsqueda típicas donde no se tiene ningún conocimiento concreto sobre los valores de las claves.

El primer paso en una búsqueda por dispersión consiste en evaluar una *función de dispersión* que transforma la clave de búsqueda en direcciones de la tabla. Idealmente, diferentes claves deben dar diferentes direcciones, pero ninguna función de dispersión es perfecta, y dos o más claves diferentes pueden dar la misma dirección de la tabla. La segunda parte de una búsqueda por dispersión es pues un proceso de *resolución de colisiones*, que permite tratar este tipo de claves. Uno de los métodos de resolución de colisiones que se estudiarán utiliza las listas enlazadas y es apropiado en situaciones muy dinámicas en las que el número de claves de búsqueda no se puede predecir. Los otros dos métodos de resolución de colisiones que se examinarán alcanzan tiempos de búsqueda muy bajos para registros almacenados en un array fijo.

La dispersión es un buen ejemplo del compromiso *espacio-tiempo*. Si no hubiera limitación de memoria, se podría hacer cualquier búsqueda con un solo acceso a la memoria, utilizando simplemente la clave como una dirección de memoria. Si no hubiera limitaciones de tiempo, se podría hacer con un mínimo de memoria utilizando un método secuencial de búsqueda. La dispersión proporciona una forma de utilizar razonablemente la memoria y el tiempo para obtener un equilibrio entre estos dos extremos. El empleo eficaz de la memoria disponible y un rápido acceso a la memoria son los objetivos básicos de cualquier método de dispersión.

La dispersión es un problema «clásico» en informática en el sentido de que los diferentes algoritmos conocidos se han estudiado con cierta profundidad y son ampliamente utilizados. Existe un gran número de justificaciones de orden empírico y analítico que apoyan la utilidad de la dispersión en una variada gama de aplicaciones.

Funciones de dispersión

El primer problema que hay que resolver es el de la realización de la función de dispersión transformando las claves en direcciones de la tabla. Éste es un problema aritmético con propiedades similares a los generadores de números aleatorios que se estudia en el Capítulo 33. Lo que se necesita es una función que transforme las claves (habitualmente enteros o cadenas cortas de caracteres) en enteros del intervalo $[0, M-1]$, donde M es el número de registros que se puede colocar en el total de memoria disponible. Una función de dispersión ideal debe ser fácil de calcular y debe ser además una aproximación a una función «aleatoria»: para cada entrada, toda salida debe ser, en cierto sentido, igualmente probable.

Como los métodos que se utilizan son aritméticos, el primer paso consiste en transformar las claves en *números* sobre los que se realicen las operaciones aritméticas. Para claves pequeñas, esto puede no significar trabajo alguno en ciertos entornos de programación, si se pueden utilizar como números las representaciones binarias de las claves (véase la presentación del comienzo del Capítulo 10). Para claves mayores, se puede intentar extraer bits de las cadenas de caracteres y empaquetarlos en una palabra en lenguaje de máquina; después se verá un método para manipular uniformemente claves de cualquier longitud.

Supóngase en primer lugar que se dispone de un gran entero que corresponde directamente a una clave. El método más comúnmente utilizado en la dispersión consiste en escoger un M primo y, para cualquier clave k , calcular $h(k) = k \bmod M$. Éste es un método directo fácil de calcular en muchos entornos de programación y dispersa las claves bastante bien.

Por ejemplo, supóngase que el tamaño de la tabla es 101 y que hay que calcular un índice para la clave de cinco caracteres C L A V E: si está codificada con el código de cinco bits utilizado en el Capítulo 10 (en el que la i -ésima letra del alfabeto se expresa por la representación binaria del número i), entonces puede verse como el número binario

0001101100000011011000101,

que es equivalente al 3540677 en base 10. Además, $3540677 \equiv 21 \pmod{101}$, así que a la clave C L A V E le corresponde («se dispersa a») la posición 21 de la tabla. Hay muchas claves posibles y relativamente pocas posiciones de la tabla, por lo que a muchas otras claves le corresponderá la misma posición (por

ejemplo, la clave A C L también tiene la dirección de dispersión 21 en el código anterior).

¿Por qué el tamaño de la tabla debe ser primo? La respuesta a esta pregunta depende de las propiedades aritméticas de la función mod. En esencia, se trata la clave como un número en base 32, a razón de un dígito por cada carácter de la clave. Se ha visto que a la clave C L A V E le corresponde el número 3540677, que también puede escribirse como

$$3 \cdot 32^4 + 12 \cdot 32^3 + 1 \cdot 32^2 + 22 \cdot 32^1 + 5 \cdot 32^0$$

puesto que C es la tercera letra del alfabeto, etc. Ahora, suponiendo que por desgracia se escoge $M = 32$: como el valor de $k \bmod 32$ no cambia al añadir múltiplos de 32, la función de dispersión de cualquier clave será simplemente ¡el valor de su último carácter! Parece natural asegurarse de que una buena función de dispersión tenga en cuenta todos los caracteres de la clave, y la forma más simple de hacerlo es eligiendo un M primo.

Pero la situación más típica es cuando las claves no son ni números ni necesariamente cortas, sino simplemente cadenas alfanuméricas (posiblemente muy largas). ¿Cómo calcular la función de dispersión de una cadena como G R A N C L A V E? En el código utilizado, a ésta le correspondería la cadena de 45 bits

001111001000001011100001101100000011011000101,

o el número

$$7 \cdot 32^8 + 18 \cdot 32^7 + 1 \cdot 32^6 + 14 \cdot 32^5 + 3 \cdot 32^4 + 12 \cdot 32^3 + 1 \cdot 32^2 + 22 \cdot 32^1 + 5,$$

que es demasiado larga para representarla con funciones aritméticas normales en la mayoría de las computadoras (habría que estar preparados para emplear claves mucho más largas). En una tal situación no se puede seguir calculando la función de dispersión como se hizo antes, transformando la clave pieza por pieza. Una vez más habrá que aprovecharse de las ventajas de las propiedades aritméticas de la función mod y de un sencillo truco de cálculo denominado el *método de Horner* (ver Capítulo 36), que se basa en escribir de otra forma el número que corresponde a la clave. En el ejemplo, se obtiene la expresión siguiente:

$$((((((7 \cdot 32 + 18)32 + 1)32 + 14)32 + 3)32 + 12)32 + 1)32 + 22)32 + 5.$$

Esto conduce a un método aritmético directo de cálculo de la función de dispersión. La implantación de este capítulo utiliza como claves cadenas y no enteros. Se supone que tipoElemento es un tipo clavecadena que permite la asignación y las operaciones de comparación por desigualdad (y por supuesto la función dispersion). Ésta es la situación más natural para describir la dispersión, aunque, por coherencia con otros capítulos, se utilicen como claves, en los

ejemplos, cadenas de un solo carácter. Esto es similar a la situación que se encuentra en los Capítulos 10 y 17 con cadenas de bits como claves: C++ permite ser explícito sobre qué operaciones se llevarán a cabo en las claves. Para la dispersión, cada tipo de clave necesita tener una función de dispersión. En las cadenas, una función de dispersión basada en el método de Horner es simplemente una forma de tratar los caracteres como dígitos.

```
unsigned clavecadena::dispersion(int M)
{
    int h; char *t = v;
    for (h = 0; *t; t++)
        h = (64*h + *t) % M;
    return h;
}
```

Aquí h es el valor de dispersión calculado y la constante 64 es, estrictamente hablando, una constante que depende de la implantación y del tamaño del alfabeto. El valor exacto de esta constante no es particularmente importante. Un inconveniente de este método es que necesita un cierto número de operaciones aritméticas para cada carácter de la clave lo cual podría ser costoso. Esto puede mejorarse procesando la clave en partes más grandes. Sin el operador $\%$, este programa calcularía el número correspondiente a la clave, como en la ecuación anterior, pero con claves muy largas el cálculo podría desbordarse. Sin embargo, con el operador $\%$ se puede calcular la función de dispersión gracias a las propiedades aditivas y multiplicativas de la operación módulo y se evita el desbordamiento porque $\%$ proporciona siempre un resultado inferior a M . La dirección calculada por el programa para G R A N C L A V E con $M = 101$ es 21.

Encadenamiento separado

Las funciones de dispersión anteriores convierten las claves en direcciones de la tabla: queda todavía por explicar cómo resolver los casos en los que dos claves dan la misma dirección. El método más directo consiste simplemente en construir, para cada dirección de la tabla, una lista enlazada con todos los registros cuyas claves se transforman en esta dirección. Puesto que las claves que tienen la misma posición en la tabla se ponen en una lista enlazada, es fácil conservarlas en orden. Esto conduce a una generalización de los métodos de búsqueda elementales que se presentaron en el Capítulo 14. Mejor que estar manteniendo una lista única con un sencillo nodo cabecera cabeza, como se sugirió entonces, es preciso mantener M listas con M nodos cabecera, inicializadas como se describe a continuación:

```

Dicc::Dicc(int tm)
{
    M = tm;
    z = new nodo; z->siguiente = z; z->info = infoNIL;
    cabezas = new nodo*[M];
    for (int i = 0; i < M; i++)
        { cabezas[i] = new nodo; cabezas[i]->siguiente = z; }
}

```

clave : **E J E M P L O D E B U S Q U E D A**

dispersión : 5 10 5 2 5 1 4 4 5 2 10 8 6 10 5 4 1

Figura 16.1 Una función de dispersión ($M = 11$).

Ahora se pueden utilizar los procedimientos del Capítulo 14 de búsqueda e inserción en una lista, modificados de tal forma que se utilice la función de dispersión para escoger entre las listas reemplazando simplemente las referencias a cabeza por $cabezas[dispersión(v)]$.

Por ejemplo, si las claves del ejemplo se insertan sucesivamente en una tabla inicialmente vacía utilizando la función de dispersión de la Figura 16.1, resulta entonces el conjunto de listas que se muestra en la Figura 16.2. Este método se denomina tradicionalmente *encadenamiento separado* porque los registros en colisión se «encadenan» juntos en listas enlazadas independientes. Las listas se pueden mantener ordenadas, pero esto no es tan importante en esta aplicación como lo fue para la búsqueda secuencial elemental porque las listas son bastante cortas. Evidentemente, el total de tiempo que se necesita para una búsqueda depende de la longitud de las listas (y de la posición relativa de las claves en ellas).

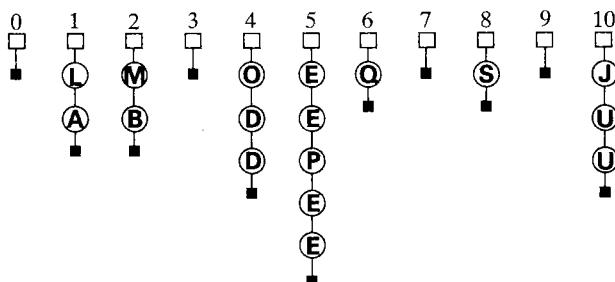


Figura 16.2 Encadenamiento separado.

Para una «búsqueda sin éxito» (la búsqueda de un registro con una clave que no está en la tabla), se puede suponer que la función de dispersión dificulta las cosas lo suficiente como para que cada una de las M listas esté en igualdad de condiciones para buscar en ella y que, al igual que en la búsqueda secuencial en una lista, cada lista en la que se busca se recorre sólo hasta la mitad (por término medio). La longitud media de la lista examinada (no contando a z) en una búsqueda sin éxito es en el ejemplo $(0+2+2+0+3+5+1+0+1+0+3)/11 \approx 1,54$. Manteniendo las listas ordenadas se podría reducir este tiempo a la mitad. Para una «búsqueda con éxito» (la búsqueda de alguno de los registros de la tabla), se supone que cada registro tiene la misma posibilidad de ser examinado: se encontrarán siete claves en primera posición de la lista, cinco en segunda, etc., por lo que la media es $(7 \cdot 1 + 5 \cdot 3 + 3 \cdot 3 + 1 \cdot 4 + 1 \cdot 5)/17 \approx 2,05$. (Este análisis supone que las claves iguales se distinguen por medio de un identificador único o de algún otro mecanismo y que la rutina de búsqueda se modifica apropiadamente para que sea capaz de buscar cualquier clave en particular.)

Propiedad 16.1 *El encadenamiento separado reduce el número de comparaciones de la búsqueda secuencial en un factor de M (por término medio), utilizando espacio extra para los M enlaces.*

Si N , el número de claves de la tabla, es mucho mayor que M , entonces una buena aproximación de la longitud media de las listas es N/M , puesto que cada uno de los M valores de dispersión es «igualmente probable» gracias al diseño de la función de dispersión. Al igual que en el Capítulo 14, las búsquedas sin éxito llegan hasta el final de una determinada lista y las búsquedas con éxito la recorren aproximadamente a la mitad.■

La implantación anterior utiliza una tabla de dispersión de enlaces a las cabeceras de las listas que contienen realmente las claves. Si no se desea mantener M nodos cabeceras de lista, una alternativa consiste en eliminarlos y hacer que cabezas sea una tabla de enlaces a las primeras claves de las listas. Esto provoca algunas complicaciones en el algoritmo. Por ejemplo, añadir un nuevo registro al comienzo de una lista se convierte en una operación diferente de la de añadir un nuevo registro en cualquier otra parte de ella, porque implica modificar una entrada de la tabla de enlaces, no un campo de un registro. Otra implantación consiste en colocar la primera clave dentro de la tabla. Aunque estas alternativas utilizan menos espacio en algunas situaciones, M es habitualmente muy pequeño en comparación con N , de modo que la comodidad añadida al utilizar nodos cabeceras de lista está casi siempre justificada.

En una implantación de encadenamiento separado, normalmente se escoge M lo suficientemente pequeño para no utilizar una gran zona de memoria contigua. Pero es probable que lo mejor sea escoger un M tal que las listas sean lo suficientemente cortas como para hacer que la búsqueda secuencial sea lo más eficaz posible: los métodos «híbridos» (como la utilización de árboles binarios

en lugar de listas enlazadas) no merecen la pena, dada su complicación. En una primera aproximación, se puede escoger un M que sea alrededor de la décima parte del número de claves que se espera que haya en la tabla, de modo que cada lista cuente con tener alrededor de diez claves. Una de las ventajas del encadenamiento separado es que este valor no es crítico: si aparecen más claves que las esperadas, entonces las búsquedas se demorarán un poco más: si hay pocas claves en la tabla, entonces puede ser que se haya utilizado un poco más de memoria de la necesaria. Si la memoria es realmente un recurso crítico, la elección de un M tan grande como se pueda permitirá aportar una ganancia de rendimiento proporcional a M .

Exploración lineal

Si el número de elementos a poner en la tabla de dispersión se puede estimar por adelantado y hay suficiente memoria contigua disponible como para tener a todas las claves y contar además con algún espacio de reserva, entonces probablemente no merezca la pena utilizar enlaces en la tabla de dispersión. Se han desarrollado varios métodos para almacenar N registros en una tabla de tamaño $M > N$, utilizando los lugares vacíos de la tabla como ayuda en la resolución de las colisiones. Tales técnicas se denominan métodos de dispersión de *direcciónamiento abierto*.

El método más simple de direcciónamiento abierto es la llamada *exploración lineal*: cuando hay una colisión (cuando la función de dispersión envía sobre un lugar de la tabla que ya está ocupado, cuya clave no es igual que la clave de búsqueda), se *explora* la siguiente posición de la tabla, comparando la clave del registro con la clave de búsqueda. Existen tres posibilidades en esta exploración: si las claves concuerdan, entonces la búsqueda termina con éxito; si allí no hay ningún registro, entonces la búsqueda termina infructuosamente; en caso contrario se explora la siguiente posición, continuando hasta que se encuentre la clave de búsqueda o una posición vacía. Si se debe insertar un registro que contiene la clave de búsqueda después de una búsqueda sin éxito, entonces simplemente se le pone en el espacio vacío de la tabla donde se terminó la búsqueda. Este método se implanta fácilmente de la forma siguiente:

```
class Dicc
{
    private:
        struct nodo
        {
            tipoElemento clave; tipoInfo info;
            nodo() { clave = " "; info = infoNIL; }
        };
        struct nodo *a;
```

```

int M;
public:
    Dicc( int tm )
        { M = tm; a = new nodo[M] }
    int buscar(tipoElemento v);
    void insertar(tipoElemento v, tipoInfo info)
    {
        int x = v.dispersion(M);
        while (a[x].info != infoNIL) x = (x+1) % M;
        a[x].clave = v; a[x].info = info;
    }
};

```

La exploración lineal necesita la existencia de una clave de valor especial para señalar las posiciones vacías de la tabla; este programa utiliza un simple espacio en blanco para este fin. El cálculo $x = (x+1) \% M$ corresponde al examen de la siguiente posición (que se pone en el comienzo cuando se alcanza el final de la tabla). Es preciso destacar que este programa no comprueba cuándo está la tabla completamente llena. (¿Qué podría pasar en este caso?) La implantación de buscar es similar a la de insertar: simplemente se añade la condición « $(v \neq a[x].clave)$ » al bucle while y se cambia la línea siguiente, la que almacena el registro, por return $a[x].info$.

clave : **E J E M P L O D E B U S Q U E D A**
 dispersión : 5 10 5 13 16 12 15 4 5 2 2 0 17 2 5 4 1

Figura 16.3 Una función de dispersión ($M = 19$).

Para el conjunto de claves del ejemplo, con $M = 19$, se tienen los valores de dispersión que se muestran en la Figura 16.3. Si estas claves se insertan en el orden dado y en una tabla inicialmente vacía, se obtiene la secuencia que se muestra en la Figura 16.4. Se observa que las claves duplicadas aparecen entre la posición inicial de la exploración y la siguiente posición vacía de la tabla, pero no necesitan ser contiguas.

El tamaño de la tabla de la exploración lineal es mayor que la del encadenamiento separado, puesto que se tiene $M > N$, pero la cantidad total de memoria que se utiliza es menor, ya que no se necesitan enlaces. El número medio de elementos que se deben examinar para una búsqueda con éxito en este ejemplo es $38/17 \approx 2.23$.

Propiedad 16.2 *Una exploración lineal utiliza menos de cinco exploraciones, por término medio, en una tabla dispersión que esté llena, al menos, en sus dos terceras partes.*

La fórmula exacta para el número medio de exploraciones necesarias, expresado en función del «factor de carga» $a = N/M$ de la tabla de dispersión, es $1/2 + 1/2(1 - a)^2$ para una búsqueda infructuosa y $1/2 + 1/2(1 - a)$ para una búsqueda con éxito. Así pues, si se toma $a = 2/3$, se obtienen cinco exploraciones en una búsqueda infructuosa y dos en una con éxito. Las búsquedas sin éxito son siempre las más costosas de las dos: una búsqueda con éxito necesitará menos de cinco exploraciones hasta que la tabla esté llena en un 90 %. A medida que la tabla se va llenando (y a se acerca a 1), estos números se van haciendo más grandes; esto no debe permitirse en la práctica, como se confirmará posteriormente.■

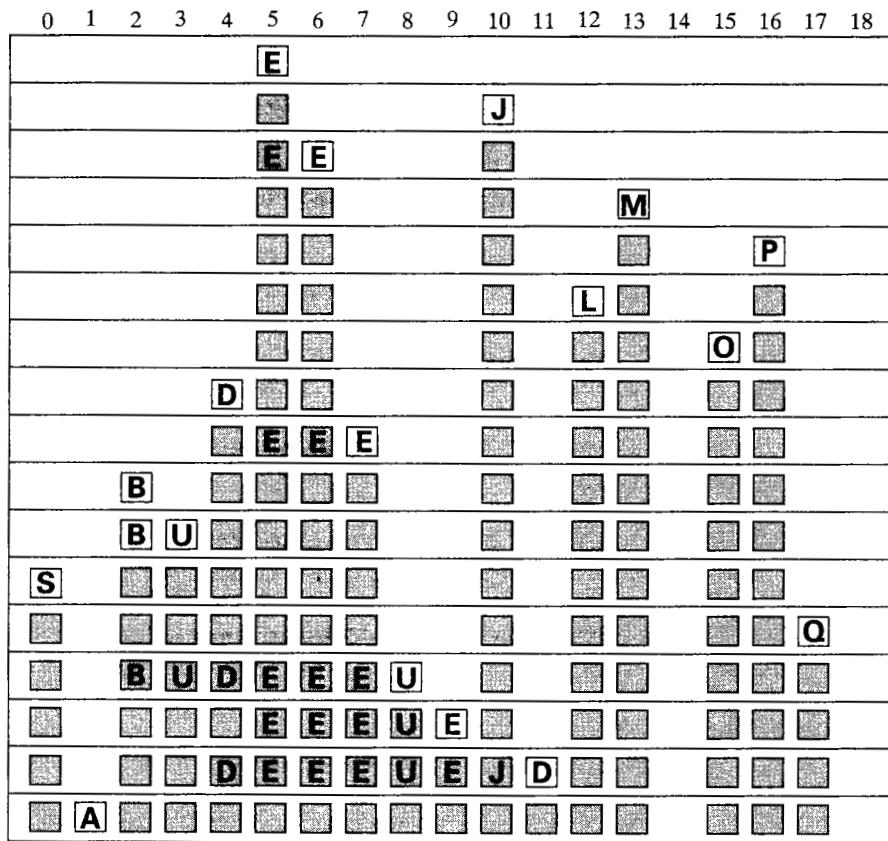


Figura 16.4 Exploración lineal.

...
...
...
...
...
...
...
...
...
...

Figura 16.5 Exploración lineal en una tabla grande.

Doble dispersión

La exploración lineal (o en su lugar cualquier método de dispersión) es válida porque garantiza que, cuando se está buscando una clave en particular, se examinan todas las claves que dan la misma dirección de tabla al aplicarles la función de dispersión (en particular la de la propia clave si está en la tabla). Desgraciadamente, en la exploración lineal se examinan también otras claves, sobre todo cuando la tabla comienza a estar muy llena: en el ejemplo anterior, la búsqueda de la segunda D implica el examen de E, U y J, ninguna de las cuales tiene el mismo valor de dispersión. Y lo que es peor, la inserción de una clave con un valor de dispersión dado puede aumentar drásticamente el tiempo de búsqueda de claves que tienen otros valores de dispersión: en el ejemplo, una inserción en la posición 18 provocaría un gran aumento del tiempo de búsqueda para la posición 17. Este fenómeno, denominado *agrupamiento*, puede hacer que la exploración lineal actúe muy lentamente en tablas casi llenas. La Figura 16.5 muestra los agrupamientos que se forman en un ejemplo de tamaño mayor.

Por fortuna, existe una forma fácil de eliminar prácticamente este problema de agrupamiento: la *doble dispersión*. La estrategia básica es la misma; la única diferencia es que, en lugar de examinar sucesivamente todas las entradas siguientes a la posición donde se ha producido la colisión, se utiliza una segunda función de dispersión para obtener un incremento fijo a utilizar en la secuencia de «exploración». Esto se implanta fácilmente insertando $u = h_2(v)$ al principio de la función y cambiando $x = (x+1) \% M$ por $x = (x+u) \% M$ dentro del bucle while.

La segunda función de dispersión se debe escoger con cuidado, ya que de otro modo el programa podría no ser válido. En primer lugar, evidentemente no se desea tener $u = 0$, puesto que conduciría a un bucle infinito en caso de colisión. En segundo lugar, es importante que M y u sean primos entre sí, para evitar que algunas de las secuencias de exploración sean muy cortas (considere el caso $M = 2u$). Esto se garantiza fácilmente haciendo a M primo y a $u < M$. En tercer lugar, la segunda función de dispersión debe ser «diferente» de la primera, pues de lo contrario puede ocurrir un agrupamiento ligeramente más complicado. Una función como $h_2(k) = M - 2 - k \bmod (M - 2)$ producirá un

clave :	E	J	E	M	P	L	O	D	E	B	U	S	Q	U	E	D	
dispersión 1 :	5	10	5	13	16	12	15	4	5	2	1	0	17	1	5	4	1
dispersión 2 :	3	6	3	3	8	4	1	4	3	6	3	5	7	3	3	4	7

Figura 16.6 Función de doble dispersión ($M = 19$).

buen surtido de «segundos» valores de dispersión, pero quizás esto sea ir demasiado lejos, ya que, especialmente para claves grandes, el coste de calcular la segunda función de dispersión prácticamente dobla el de la búsqueda, para evitar solamente algunas exploraciones para eliminar el agrupamiento. En la práctica puede ser suficiente una segunda función de dispersión mucho más simple, como por ejemplo $h_2(k) = 8 - (k \bmod 8)$. Esta función sólo utiliza los últimos tres bits de k ; puede ser apropiado utilizar un mayor número de bits para una tabla más grande, aunque el efecto, incluso si es notorio, no es probable que sea significativo en la práctica.

Para las claves del ejemplo, estas funciones producen los valores de dispersión que se muestran en la Figura 16.6. La Figura 16.7 muestra la tabla que se obtiene por la inserción sucesiva de estas claves en una tabla inicialmente vacía y utilizando la doble dispersión con estos valores.

El número medio de elementos examinados en una búsqueda con éxito es ligeramente superior al de la exploración lineal para el mismo ejemplo: $34/17 = 2$. Pero en una tabla más espaciada, hay muchos menos agrupamientos, tal como se muestra en la Figura 16.8. En este ejemplo, hay dos veces menos agrupamientos que en la exploración lineal (Figura 16.5), o, de forma equivalente, el agrupamiento medio es dos veces más pequeño.

Propiedad 16.3 *La doble dispersión utiliza menos exploraciones, por término medio, que la exploración lineal.*

La fórmula exacta para el número medio de exploraciones que se hacen en la técnica de doble dispersión con una función de doble dispersión «independiente» es $1/(1 - a)$ para una búsqueda sin éxito y $-\ln(1 - a)/a$ para una búsqueda con éxito. (Estas fórmulas son el resultado de un análisis matemático profundo y aún no han sido verificadas para un a muy grande.) La segunda (y más sencilla) de las dos funciones de dispersión de las antes recomendadas no cumple con esto exactamente, pero puede ser válida, en especial si se utilizan los suficientes bits para hacer que el número de valores posibles esté próximo a M . En la práctica, esto significa que con la doble dispersión se puede utilizar una tabla más pequeña para lograr los mismos tiempos de búsqueda que con la exploración lineal: el número medio de exploraciones es inferior a cinco, en una búsqueda sin éxito, si la tabla está llena a menos del 80 %, y para una búsqueda con éxito si lo está a menos del 99 %.■

Los métodos de direccionamiento abierto pueden no ser convenientes en una

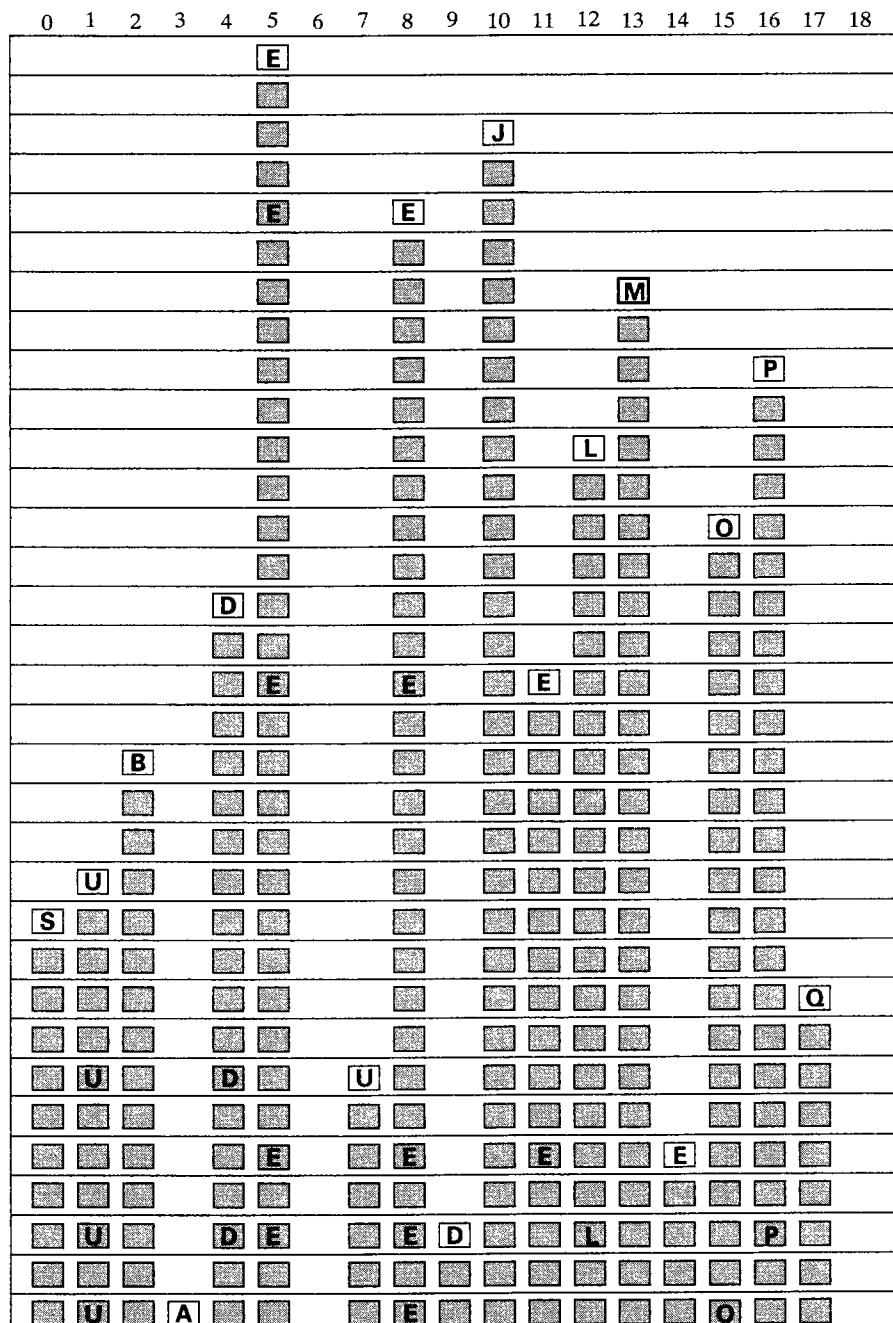


Figura 16.7 Doble dispersión.

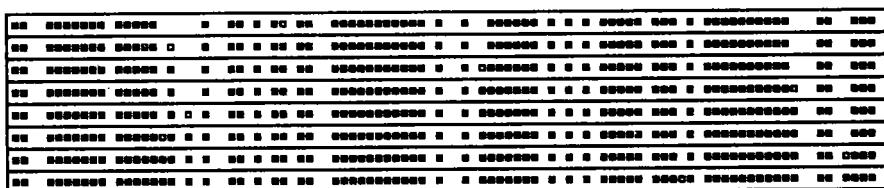


Figura 16.8 Doble dispersión en una tabla más grande.

situación dinámica cuando se tiene que procesar un número imprevisible de inserciones y eliminaciones. En primer lugar ¿cuál debe ser el tamaño de la tabla? De una forma u otra se deben hacer estimaciones de cuántas inserciones se esperan, pero el rendimiento se degrada rápidamente a medida que la tabla comienza a llenarse. Una solución común para este problema es hacer una redispersión en una tabla más grande, de la forma menos frecuente que sea posible. En segundo lugar hay que tener precaución con la eliminación: un registro no se puede eliminar tranquilamente de una tabla construida por medio de una exploración lineal o de doble dispersión. La razón es que las últimas inserciones en la tabla puede que hayan saltado la posición de este registro y, una vez eliminado, las búsquedas terminarán en el hueco dejado por el registro eliminado. Un medio de resolver este problema es tener otra clave especial que pueda servir de comodín para las búsquedas, pero que pueda ser identificada y recordada como una posición vacía para las inserciones. Se observa que ni el tamaño de la tabla ni la supresión de elementos presentan problemas particulares en el caso del encadenamiento separado.

Perspectiva

Los métodos presentados en lo anterior han sido analizados completamente y es posible comparar su rendimiento con algún detalle. Las fórmulas proporcionadas en el texto son la síntesis de los análisis detallados descritos por D. E. Knuth en su libro sobre ordenación y búsqueda. Dichas fórmulas indican cómo se degrada el rendimiento en el direccionamiento abierto cuando α tiende a 1. Para M y N grandes, con una tabla llena en un 90 %, la exploración lineal necesitará alrededor de 50 exploraciones para una búsqueda sin éxito, en comparación con las 10 de la doble dispersión. Pero en la práctica, no se debe dejar jamás que una tabla de dispersión ¡llegue a llenarse en un 90%! Para pequeños valores del factor de carga, sólo serán necesarias algunas exploraciones; si no es posible lograr factores de carga pequeños, no se debe utilizar la técnica de dispersión.

La comparación de la exploración lineal y la doble dispersión con el encadenamiento separado es algo más complicada, ya que se dispone de menos me-

moria en el método del direccionamiento abierto (puesto que no hay enlaces). El valor de a debe modificarse, para tener esto en cuenta, en función del tamaño relativo de las claves y los enlaces. Esto significa que normalmente no será justificable la elección del encadenamiento separado en lugar de la doble dispersión, por criterios de rendimiento.

La selección del mejor método de dispersión para una aplicación determinada puede resultar muy difícil. Sin embargo, en una situación dada raramente se necesita el mejor método y las distintas técnicas suelen tener características de comportamiento similares mientras que los recursos de memoria no se fueren demasiado. En general, la mejor elección consiste en utilizar el método de encadenamiento separado para reducir drásticamente el tiempo de búsqueda cuando no se conoce por adelantado el número de registros a procesar (y se dispone de un buen administrador de memoria) y utilizar la doble dispersión para buscar en conjuntos de claves cuyo tamaño aproximado se puede predecir.

Se han desarrollado muchos otros métodos de dispersión que tienen aplicación en situaciones especiales. Aunque no se puede entrar en detalles, se exponen brevemente dos ejemplos para ilustrar la naturaleza de los métodos de dispersión especializados. Éstos, y muchos otros métodos, se describen completamente en los libros de Knuth y Gonnet.

El primero de ellos, denominado *dispersión ordenada*, explota el orden de una tabla de direccionamiento abierto. En la exploración lineal estándar, se detiene la búsqueda cuando se encuentra una posición vacía de la tabla o un registro con una clave igual a la de búsqueda; en la dispersión ordenada, se detiene la búsqueda cuando se encuentra un registro con una clave mayor o igual que la clave de búsqueda (la tabla se debe haber construido hábilmente para hacer este trabajo). Este método reduce el tiempo de una búsqueda infructuosa aproximadamente al mismo de una con éxito. (Éste es el mismo tipo de mejora que se hace en el encadenamiento separado.) Este método es útil para aplicaciones donde la búsqueda infructuosa sucede frecuentemente. Por ejemplo, un sistema de tratamiento de texto puede tener un algoritmo para separar las palabras que funcione bien para la mayoría de las palabras, pero no para algunos casos excepcionales (como «excepción»). Esta situación podría arreglarse buscando todas las palabras en un diccionario de *excepciones* relativamente pequeño, que contenga aquellas que deben utilizarse de forma especial, y así la mayor parte de las búsquedas serán infructuosas.

De forma similar, existen métodos para desplazar algunos registros durante una búsqueda sin éxito con el fin de hacer que las búsquedas con éxito sean más eficaces. De hecho, R. P. Brent desarrolló un método en el que el tiempo medio de búsqueda puede ser acotado por una constante, que es muy útil en aplicaciones que implican frecuentes búsquedas con éxito en tablas muy grandes, tales como los diccionarios.

Éstos son sólo dos ejemplos de un gran número de mejoras que se han propuesto para la dispersión. Muchas de estas mejoras son interesantes y tienen importantes aplicaciones. Sin embargo, se debe tener mucha precaución en la utilización prematura de métodos avanzados, excepto por expertos que tengan

serios problemas en aplicaciones de búsqueda, porque, en definitiva, el encadenamiento separado y la doble dispersión son simples, eficaces y bastante aceptables en la mayoría de las aplicaciones.

En muchas aplicaciones es preferible la dispersión a las estructuras de árboles binarios, porque es más simple y ofrece tiempos de búsqueda muy rápidos (constantes), si hay espacio disponible para tablas lo suficientemente grandes. Las estructuras de árboles binarios tienen la ventaja de ser dinámicas (no se necesita información previa sobre el número de inserciones), pueden garantizar el rendimiento en el peor caso (todos los elementos se pueden colocar en el mismo lugar al igual que lo podría hacer el mejor método de dispersión) y permiten una gran variedad de operaciones (la más importante, la función *ordenar*). Cuando estos factores no son importantes, la dispersión es ciertamente el método de búsqueda ideal.

Ejercicios

1. Describir cómo podría implementarse una función de dispersión haciendo uso de un buen generador de números aleatorios. ¿Tendría sentido implementar un generador de números aleatorios utilizando una función de dispersión?
2. ¿Cuánto tiempo haría falta en el peor caso para insertar N claves en una tabla inicialmente vacía, utilizando el método de encadenamiento separado con listas desordenadas? Responder a la misma pregunta pero con listas ordenadas.
3. Dar el contenido de la tabla de dispersión que resulta cuando se insertan las claves C U E S T I O N F A C I L en una tabla inicialmente vacía de tamaño 13 utilizando la exploración lineal. (Utilizar $h_1(k) = k \bmod 13$ para la función de dispersión de la k -ésima letra del alfabeto.)
4. Dar el contenido de la tabla de dispersión que resulta cuando se insertan las claves C U E S T I O N F A C I L en una tabla inicialmente vacía de tamaño 13 utilizando la doble dispersión. (Utilizar el $h_1(k)$ de la pregunta anterior y $h_2(k) = 1 + (k \bmod 11)$ para la segunda función de dispersión.)
5. Aproximadamente, ¿cuántas exploraciones deben hacerse cuando se utiliza la doble dispersión para construir una tabla de N claves iguales?
6. ¿Qué método de dispersión utilizaría para una aplicación en la que es posible que estén presentes muchas claves iguales?
7. Suponiendo que el número de elementos a insertar en una tabla de dispersión se conoce por adelantado. ¿Bajo qué condiciones es preferible el método del encadenamiento separado a la doble dispersión?
8. Suponiendo que un programador tiene un error en un programa de doble dispersión de modo que una de las dos funciones devuelve siempre el mismo valor (distinto de 0), describir lo que sucede en cada situación (cuando es la primera función la que está mal y cuando lo es la segunda).

9. ¿Qué función de dispersión debe utilizarse si se conoce por adelantado que los valores de las claves pertenecen a un intervalo relativamente pequeño?
10. Hacer una crítica del algoritmo siguiente para suprimir en una tabla de dispersión construida por el método de exploración lineal. Explorar hacia la derecha, desde el elemento que se va a eliminar (dando la vuelta si es necesario) hasta encontrar una posición vacía, después explorar hacia la izquierda hasta encontrar un elemento con el mismo valor de dispersión. Finalmente reemplazar el elemento a suprimir por este último dejando vacía su posición en la tabla.

Búsqueda por residuos

Varios métodos de búsqueda producen examinando las claves de búsqueda a razón de un bit cada vez, en lugar de hacer comparaciones completas entre claves en cada paso. Estos métodos, denominados *métodos de búsqueda por residuos*, trabajan con los bits de las propias claves, y no con las versiones transformadas de las claves utilizadas en la dispersión. Al igual que los de ordenación por residuos (ver el Capítulo 10), estos métodos pueden ser muy útiles cuando los bits de las claves de búsqueda son fácilmente manipulables y los valores de las claves están bien distribuidos.

La ventaja principal de los métodos de búsqueda por residuos es que proporcionan un rendimiento razonable en el peor caso, sin las complicaciones de los árboles equilibrados; también proporcionan un método fácil para utilizar claves de longitud variable; algunos permiten incluso ganar espacio almacenando parte de la clave dentro de la estructura de búsqueda; y, finalmente, permiten un acceso muy rápido a los datos, compitiendo tanto con los árboles binarios de búsqueda como con la dispersión. Sus inconvenientes son que toda inclinación en los datos puede provocar un mal rendimiento por degeneración de los árboles (y todo dato compuesto por caracteres está inclinado) y que algunos de estos métodos pueden malgastar inútilmente el espacio de la memoria. Al igual que con la ordenación por residuos, estos métodos se diseñan para aprovechar las características particulares de las arquitecturas de las computadoras: puesto que utilizan las propiedades digitales de las claves, es difícil, o imposible, hacer implementaciones eficaces en algunos lenguajes de alto nivel.

En este capítulo se examinarán una serie de métodos, de los que cada uno corrige un defecto inherente al anterior, y se terminará con un método importante que es bastante útil en aplicaciones de búsqueda en las que se trabaja con claves de gran longitud. Además se estudiará el análogo de la «ordenación en tiempo lineal» del Capítulo 10, una búsqueda en «tiempo constante» basada en el mismo principio.

Árboles de búsqueda digital

El método de búsqueda por residuos más simple es el de búsqueda digital: el algoritmo es precisamente el mismo que el de búsqueda por árbol binario, excepto que el movimiento por las ramas del árbol no se hace de acuerdo con el resultado de una comparación entre claves, sino con los bits de la clave. En el primer nivel se utiliza el primer bit, en el segundo nivel se utiliza el segundo bit, y así hasta encontrar un nodo externo. El código es virtualmente el mismo que el de la búsqueda por árbol binario. La única diferencia es que las claves son del tipo `clavebits` utilizado en la ordenación por residuos y se utiliza la función `bits`, para tener acceso a los bits individuales, en lugar de las comparaciones entre claves. (Se recuerda del Capítulo 10 que `v.bits(k,j)` son los `j` bits que aparecen a `k` bits de distancia del extremo derecho de la representación binaria de `v`; esto se puede implementar eficazmente en lenguaje de máquina desplazando `k` bits hacia la derecha, y poniendo después a 0 todos los bits menos los `j` más a la derecha.)

```
TipoInfo Dicc::buscar(tipoElemento v) //Arbol digital
{
    struct nodo *x = cabeza;
    int b = tipoElemento::maxb;
    z->clave = v;
    while (v != x->clave)
        x = (v.bits(b--, 1)) ? x->der : x->tizq;
    return x->info;
}
```

Las estructuras de datos de este programa son las mismas que las que se utilizaron en los árboles binarios de búsqueda elementales. La constante `maxb` es el número de bits de las claves que se van a ordenar. El programa supone que el primer bit de cada clave (el que está a $(\text{maxb}+1)$ de la derecha) es 0 (tal vez la clave sea el resultado de utilizar `bits` con `maxb` como segundo argumento), así que la búsqueda comienza en `cabeza`, un enlace a un nodo de cabecera del árbol con clave 0, que posee un enlace izquierdo que apunta al árbol de búsqueda. Así el procedimiento de inicialización para este programa es el mismo que para el de búsqueda por árbol binario, excepto que se empieza con `cabeza->izq = z` en lugar de `cabeza->der = z`.

En el Capítulo 10 se vio que las claves iguales son un anatema en la ordenación por residuos: lo mismo sucede en la búsqueda por residuos, no en este algoritmo en particular, pero sí en los que se examinarán más adelante. Así pues en este capítulo se supone que todas las claves que aparecen en la estructura de datos son distintas: si es necesario, se puede mantener una lista enlazada, para cada valor de clave, de todos los registros cuyas claves tienen ese valor. Como

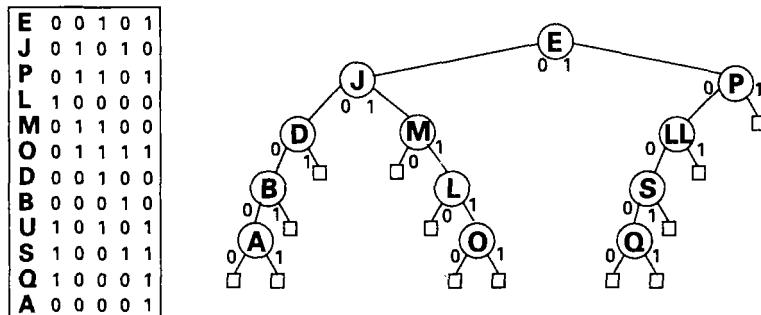


Figura 17.1 Un árbol de búsqueda digital.

en los capítulos precedentes, se supone que la i -ésima letra del alfabeto se representa por medio de la representación binaria de cinco bits de i . Las claves del ejemplo que se utilizarán en este capítulo se muestran en la Figura 17.1. Para ser consistentes con bits, se considera que los bits están numerados del 0 al 4 y de derecha a izquierda. Así por ejemplo, el bit 1 es el único bit 1 (no cero) de B y el bit 4 es el único bit 1 (no cero) de P.

El procedimiento de inserción para árboles de búsqueda digital se obtiene directamente del procedimiento correspondiente para los árboles binarios de búsqueda:

```
void Dicc::insertar(tipoElemento v, tipoInfo info)
{
    struct nodo *p, *x = cabeza;
    int b = tipoElemento::maxb;
    while (x != z)
    {
        p = x;
        x = (v.bits(b--, 1)) ? x->der : x->izq;
    }
    x = new nodo;
    x->clave = v; x->info = info; x->izq = z; x->der = z;
    if (v.bits(b+1, 1)) p->der = x; else p->izq = x;
}
```

El árbol construido por este programa, cuando las claves del ejemplo se insertan en un árbol inicialmente vacío, se muestra en la Figura 17.1. La Figura 17.2 muestra lo que sucede al añadir una nueva clave Z = 11010 al árbol de la Figura 17.1. Se debe ir por la derecha dos veces, porque los dos primeros bits de Z

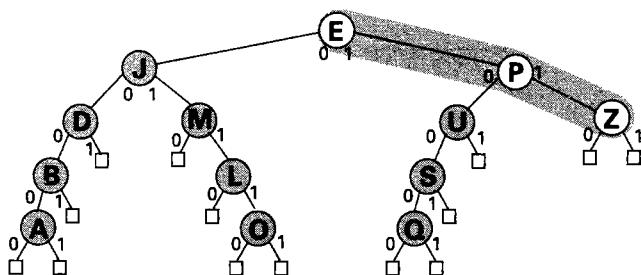


Figura 17.2 Inserción (de Z) en un árbol de búsqueda digital.

son 1, hasta donde se encuentra el nodo externo a la derecha de P, que es donde se insertará Z.

El peor caso para árboles construidos con búsqueda digital es mucho mejor que el de los árboles binarios de búsqueda, si el número de claves es grande y no son largas. La longitud del camino más largo en un árbol de búsqueda digital es el mayor número de bits sucesivos iguales de dos claves cualesquiera del árbol, a partir del bit más a la izquierda, y esta cantidad es relativamente pequeña en muchas aplicaciones (por ejemplo, si las claves están compuestas de bits aleatorios).

Propiedad 17.1 *Una búsqueda o inserción en un árbol de búsqueda digital, construido sobre N claves de b bits aleatorios, necesita alrededor de $\lg N$ comparaciones por término medio y b comparaciones en el peor caso.*

Es evidente que ningún camino será nunca más largo que el número de bits de las claves: por ejemplo, un árbol de búsqueda digital construido a partir de claves de ocho caracteres, con seis bits por carácter, no tendrá ningún camino mayor que 48, incluso si hay cientos de miles de claves. Demostrar que los árboles de búsqueda digital están casi perfectamente equilibrados necesita un análisis que va más allá del alcance de este libro, aunque este hecho confirma la noción intuitiva de que el «siguiente» bit de una clave aleatoria tiene la misma probabilidad de comenzar por 0 que por 1, así que la mitad de las claves deben encontrarse a cada lado de un nodo dado. La Figura 17.3 muestra un árbol de búsqueda digital construido a partir de 95 claves aleatorias de 7 bits. Este árbol está bastante bien equilibrado.■

Así, los árboles de búsqueda digital ofrecen una alternativa atractiva a los de búsqueda binaria estándar, *siempre y cuando* la extracción de bits sea tan fácil de hacer como la comparación entre claves (consideración que es dependiente de la máquina).

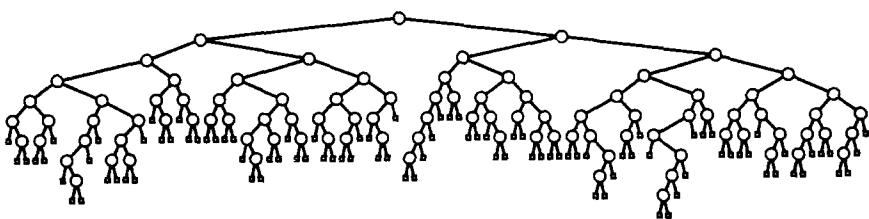


Figura 17.3 Un gran árbol de búsqueda digital.

Árboles de búsqueda por residuos

Es bastante frecuente que las claves de búsqueda sean muy largas y estén constituidas, quizás, por veinte caracteres o más. En tal situación, el coste de la comparación de la igualdad entre la clave de búsqueda y una clave de la estructura de datos puede ser preponderante y no debe ignorarse. La búsqueda por árbol digital efectúa una tal comparación en cada nodo del árbol; en esta sección se verá que en la mayor parte de los casos es posible lograrlo con una sola comparación por búsqueda.

La idea es no almacenar claves en los nodos internos del árbol, sino poner todas en los nodos externos. Esto es, en lugar de utilizar z para los nodos terminales de la estructura, se ponen nodos que contienen las claves de búsqueda. Así pues, se tienen dos tipos de nodos en la estructura: nodos internos, que sólo contienen enlaces a otros nodos, y nodos terminales que contienen claves y no enlaces. (Fredkin denominó a este método *trie* porque es útil para la extracción («*retrieval*»), palabra que suele pronunciarse «*trai-i*» o simplemente «*trai*»). Para buscar una clave en una estructura como ésta, es preciso moverse por las ramas de acuerdo con sus bits, al igual que anteriormente, pero sin comparar la clave con nada hasta que no se alcance un nodo externo. Cada clave del árbol se almacena en un nodo terminal del camino descrito por el conjunto de los primeros bits de la clave y, como cada clave de búsqueda termina en un nodo terminal, se necesita una comparación completa de las claves para terminar la búsqueda.

La Figura 17.4 muestra el método *trie* de búsqueda por residuos para las claves E J M P L. Por ejemplo, para llegar a E, se parte de la raíz yendo primero a la izquierda y luego otra vez a la izquierda, ya que los dos primeros bits de E son 00; pero al contrario, como ninguna de las claves del método *trie* comienza con los bits 11, al moverse en la dirección derecha-derecha se llega a un nodo terminal. Antes de pensar en una inserción, el lector debe reflexionar sobre la sorprendente propiedad de que la estructura *trie* es independiente del orden en el que se insertan las claves: hay un único *trie* para cualquier conjunto dado de claves distintas.

Como es habitual, después de una búsqueda sin éxito, se puede insertar la

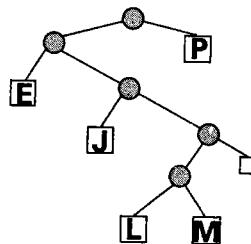


Figura 17.4 Un trie de árbol de búsqueda por residuos.

clave reemplazando el nodo terminal en el que terminó la búsqueda, *a condición* de que no contenga una clave. Éste es el caso cuando se inserta O en el trie de la Figura 17.4, como se muestra en el primer trie de la Figura 17.5. Si el nodo externo en el que termina la búsqueda contiene una clave, debe reemplazarse por un nodo interno que contenga en los nodos externos por debajo de él, la clave en cuestión y la clave en la que termina la búsqueda. Por desgracia, si estas claves coinciden en más posiciones de bits, es necesario añadir algunos nodos terminales que no corresponden a claves del árbol (es decir, nodos internos con un nodo terminal vacío como hijo). Esto es lo que sucede cuando se inserta D, como se muestra en el segundo trie de la Figura 17.5. El resto de la Figura 17.5 muestra cómo se completa el ejemplo cuando se añaden las claves B U S Q A.

La implementación de este método en C++ requiere algunos trucos por la necesidad de mantener dos tipos de nodos sobre cada uno de los cuales podrían apuntar los enlaces de los nodos internos. Éste es un ejemplo de un algoritmo para el que una implementación de bajo nivel puede resultar más simple que una representación de alto nivel. Se omite el código para este caso porque posteriormente se verá una mejora que evita este problema.

El subárbol izquierdo de un trie de búsqueda por residuos contiene todas las claves que tienen un 0 como bit más significativo y el subárbol derecho contiene todas las claves que tienen un 1 como bit más significativo. Esto conduce a una correspondencia inmediata con la ordenación por residuos: la búsqueda por trie binario partitiona el archivo exactamente de la misma forma que la ordenación por intercambio de residuos. (Se puede comparar el trie anterior con la Figura 10.1, el diagrama de partición de la ordenación por intercambio de residuos, teniendo en cuenta que las claves son ligeramente diferentes.) Esta correspondencia es análoga a la que existe entre la búsqueda por árbol binario y el Quicksort.

Propiedad 17.2 Una búsqueda o una inserción en un trie de búsqueda por residuos necesita alrededor de $\lg N$ comparaciones de bits por término medio y b comparaciones de bits en el peor caso, en un árbol construido a partir de N claves aleatorias de b bits.

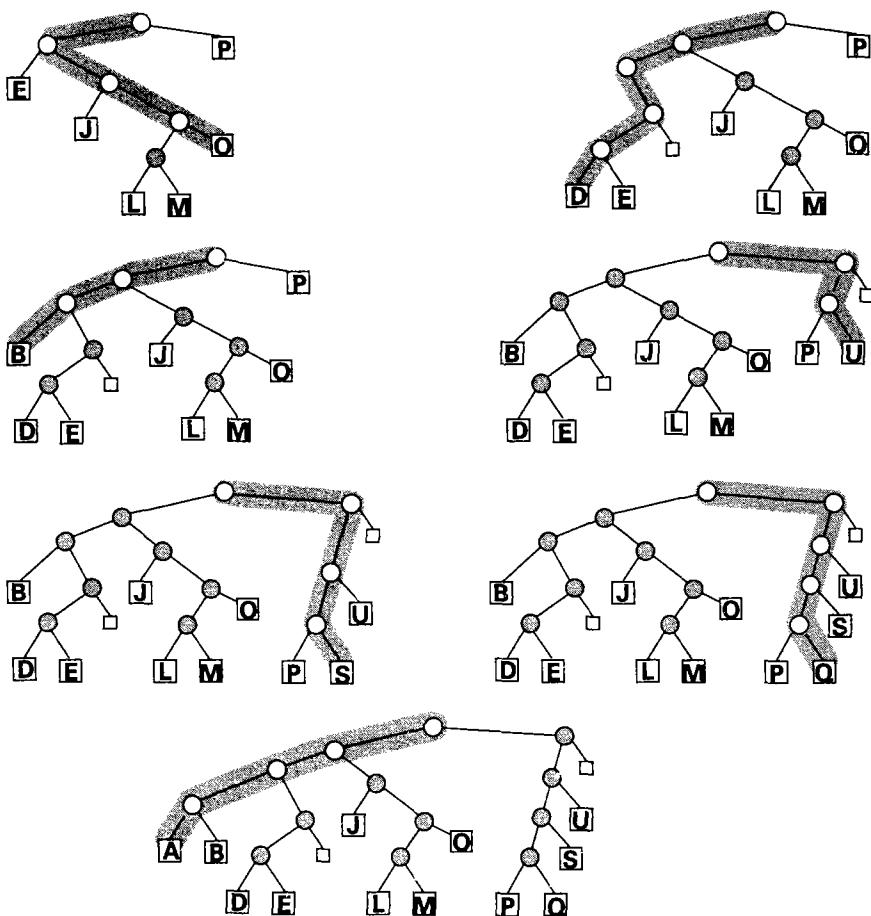


Figura 17.5 Construcción de un trie de búsqueda por residuos.

Como se hizo anteriormente, el resultado del peor caso se obtiene directamente del algoritmo y para el caso medio se requiere un análisis matemático que sobrepasa el alcance de este libro, aunque esta propiedad da validez a la noción intuitiva de que cada bit examinado puede ser lo mismo un 0 que un 1, así que aproximadamente la mitad de las claves deben encontrarse en cada lado de un nodo del trie.■

Una característica molesta de los tries por residuos, que los distingue de los otros tipos de árboles de búsqueda que se han visto, es la ramificación «unidireccional» que se necesita para las claves con un gran número de bits iguales. Por ejemplo, las claves que difieren solamente en el último bit necesitan un camino cuya longitud sea igual a la de la clave, independientemente del número

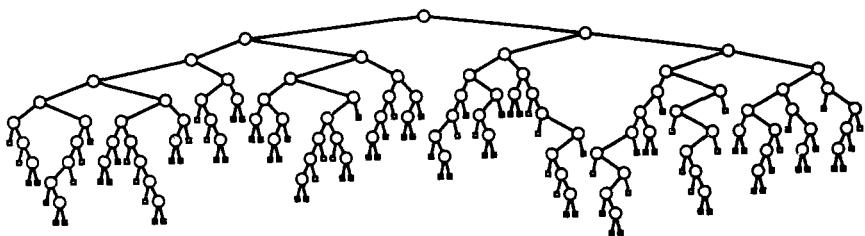


Figura 17.6 Un gran trie de búsqueda por residuos.

de claves que haya en el árbol. El número de nodos internos puede ser algo mayor que el de claves.

Propiedad 17.3 *Un trie de búsqueda por residuos construido a partir de N claves aleatorias de b bits contiene alrededor de $N/\ln 2 \approx 1.44N$ nodos de media.*

Una vez más, la demostración de esta afirmación va más allá del alcance de este libro, aunque se puede verificar empíricamente. La Figura 17.6 muestra un trie construido a partir de 95 claves aleatorias de 10 bits, que tiene 131 nodos. ■

La altura de los tries se mantiene limitada por el número de bits de las claves, pero se podría considerar la posibilidad de procesar registros con claves muy largas (1.000 bits o más) que quizás presenten cierta uniformidad, como puede ser el caso de datos codificados por caracteres. Una forma de acortar los caminos de los árboles es utilizar mucho más de dos enlaces por nodo (aunque esto puede agudizar el problema de «espacio» al utilizar demasiados nodos); otra forma es «colapsar» los caminos que contienen ramas unidireccionales en enlaces sencillos. Estos métodos se presentarán en las dos secciones siguientes.

Búsqueda por residuos múltiple

En la ordenación por residuos se vio que se pueden obtener importantes mejoras en la velocidad considerando varios bits a la vez. Esto es también cierto en la búsqueda por residuos: examinando m bits a la vez, se puede aumentar la velocidad en un factor 2^m . Sin embargo, hay una situación que impone algo más de cuidado al aplicar esta idea, lo que no fue necesario en la ordenación por residuos. El problema es que considerar m bits a la vez implica utilizar nodos con $M = 2^m$ enlaces, lo que puede conducir a derrochar un volumen considerable de espacio por los enlaces no utilizados.

Por ejemplo, si $M = 4$ el trie que se obtiene para las claves del ejemplo es el que se muestra en la Figura 17.7. Para buscar en este trie, se consideran los bits

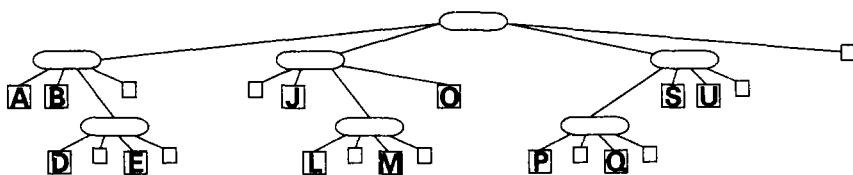


Figura 17.7 Un trie por residuos de 4 vías.

de dos en dos: si los dos primeros bits son 00, entonces se sigue el enlace izquierdo del primer nodo; si son 01, se sigue el segundo enlace; si son 10, se sigue el tercer enlace, y si son 11, el enlace derecho. La dirección para moverse por las ramas hacia el siguiente nivel se obtiene de acuerdo con el tercero y cuarto bits, etc. Por ejemplo, para buscar $V = 10.110$ en el trie de la Figura 17.7 se sigue el tercer enlace de la raíz y luego el cuarto enlace del tercer hijo de la raíz, para acceder a un nodo terminal, de modo que la búsqueda no tiene éxito. Para insertar V , se debe reemplazar dicho nodo por uno nuevo que contenga a V (y cuatro enlaces externos).

Es de destacar que hay un cierto despilfarro de espacio en este árbol por el gran número de enlaces a nodos terminales que no se utilizan. A medida que M crece este efecto se acentúa: esto conduce a que el número de enlaces utilizados sea de alrededor de $MN/\ln M$ para claves aleatorias. Por otra parte, éste es un método de búsqueda muy eficaz: el tiempo de ejecución es de alrededor de $\log_M N$. Se puede establecer un compromiso razonable entre la eficacia en tiempo de los tries múltiples y la economía de espacio de otros métodos, utilizando un método «híbrido» con un valor muy grande de M en lo alto (por ejemplo en los dos primeros niveles) y un valor pequeño de M (o algún método elemental) en los niveles inferiores. Aquí, otra vez, las implementaciones eficaces de tales métodos pueden ser muy complicadas, debido a la presencia de múltiples tipos de nodos.

Por ejemplo, un árbol de dos niveles y 32 vías divide a las claves en 1.024 categorías, cada una accesible por medio de un descenso de dos niveles del árbol. Esto sería bastante útil en archivos de miles de claves, porque posiblemente existen (sólo) unas pocas claves por categoría. Por otro lado, un M pequeño sería apropiado para archivos de cientos de claves, porque de lo contrario la mayoría de las categorías estarían vacías y se gastaría demasiado espacio, y un M grande sería apropiado para archivos con millones de claves, porque de lo contrario la mayoría de las categorías tendrían muchas claves y se perdería mucho tiempo en las búsquedas.

Es asombroso comprobar que la búsqueda «híbrida» corresponde muy de cerca a la forma en que los humanos buscan las cosas, por ejemplo, los nombres en una guía de teléfonos. El primer paso es una decisión múltiple («Vamos a ver, comienza con 'A'»), seguida probablemente de alguna decisión de dos vías («Está antes de 'Andrés', pero después de 'Aivar'») y después de una búsqueda

secuencial («‘Alfonso’... ‘Algora’... ‘Algrano’... ¡No, ‘Algoritmos’ no está en la lista!»). Por supuesto, las computadoras están posiblemente mejor dotadas que los humanos para las búsquedas múltiples, así que son suficientes dos niveles. También las ramificaciones de 26 vías (incluso con más niveles) son una alternativa bastante razonable a considerar para claves que estén compuestas solamente por letras (por ejemplo, en un diccionario).

En el próximo capítulo, se verá un método sistemático para adaptar la estructura con el fin de obtener provecho de la búsqueda por residuos múltiples en el caso de archivos de tamaño arbitrario.

Patricia

Como se ha señalado, el método de búsqueda trie por residuos tiene dos defectos molestos: la «ramificación de una sola vía», que provoca la creación de nodos extra en el árbol, y la existencia de dos tipos diferentes de nodos, lo que complica en cierta forma el código (en especial el de una inserción). D.R. Morrison descubrió una forma de evitar ambos problemas en un método que denominó *Patricia* («*Practical Algorithm To Retrieve Information Coded In Alphanumeric*» o «Algoritmo Práctico para Recuperar Información Codificada en Alfanumérico»). El algoritmo que se da a continuación no es exactamente de la misma forma que la presentada por Morrison, porque él estaba interesado en aplicaciones de «búsqueda de cadenas» del tipo de las que se verán en el Capítulo 19. En el contexto presente, Patricia permite la búsqueda de N claves de longitud arbitraria en un árbol que tiene exactamente N nodos, necesitando sólo una comparación completa por búsqueda.

La ramificación unidireccional se evita gracias a un simple recurso: cada nodo contiene el índice del bit que se deberá comprobar para decidir qué camino tomar cuando se salga de este nodo. Se evitan los nodos terminales reemplazando los enlaces hacia los nodos externos por enlaces que apuntan hacia niveles superiores del árbol, lo que lleva a la noción habitual de los nodos normales del árbol, con una clave y dos enlaces. Pero en Patricia, las claves de los nodos no se utilizan para controlar la búsqueda en el recorrido hacia abajo del árbol; simplemente se almacenan como referencia para cuando se alcance el fondo del árbol. Para ver cómo funciona Patricia, se comienza por observar cómo opera en un árbol tipo y luego examinar cómo se construye el árbol. El árbol Patricia que se muestra en la Figura 17.8 se construyó insertando sucesivamente las claves del ejemplo.

Para buscar en este árbol, se comienza por la raíz y se desciende utilizando el índice de bit de cada nodo para saber qué bit examinar en la clave de búsqueda. Se sigue hacia la derecha si ese bit es 1 y hacia la izquierda si es 0. Las claves de los nodos no se examinan del todo en ese descenso. Tarde o temprano se encuentra un enlace ascendente: cada uno de ellos apunta hacia la única clave del árbol que contiene los bits que causarían una búsqueda que alcance tal en-

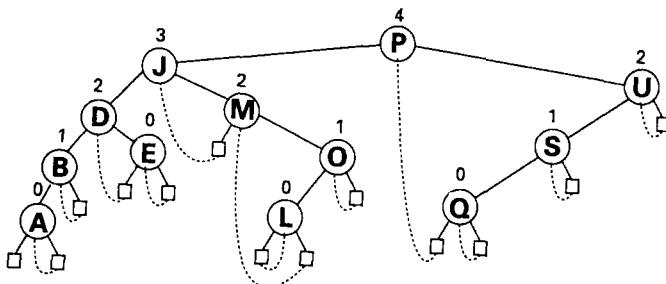


Figura 17.8 Un árbol Patricia.

lace. Por ejemplo, S es la única clave del árbol que concuerda con el patrón de bits 10*11. Así que si la clave del nodo apuntado por el primer enlace ascendente que se encuentre es igual a la clave buscada, la búsqueda tiene éxito, y en caso contrario será infructuosa. En los tries, todas las búsquedas terminan en nodos externos, así que se necesita una comparación completa de la clave para determinar cuándo una búsqueda tuvo éxito o no; para Patricia todas las búsquedas terminan en enlaces ascendentes, así que también se necesita una comparación completa de la clave para determinar si la búsqueda tuvo éxito o no. Además, es fácil verificar si un enlace es ascendente o no, porque los índices de bits de los nodos (por definición) decrecen a medida que se desciende por el árbol. Esto conduce al siguiente código de búsqueda para Patricia, que es tan simple como el del árbol por residuos o el de la búsqueda por trie:

```
tipoInfo Dicc::buscar(tipoElemento v) // Arbol Patricia
{
    struct nodo *p, *x;
    p = cabeza; x = cabeza->izq;
    while (p->b > b)
    {
        p = x;
        x = (bits(v, x->b, 1)) ? x->der : x->izq;
    }
    if (v != x->clave) return infoNIL;
    return x->info;
}
```

Esta función encuentra el único nodo que podría contener el registro de clave v, y después verifica si la búsqueda ha tenido éxito o no. Así, para buscar Z=11010 en el árbol anterior, se sigue hacia la derecha, luego hacia la izquierda y después otra vez a la derecha, encontrando el enlace ascendente que lleva a S; así que la búsqueda no tiene éxito.

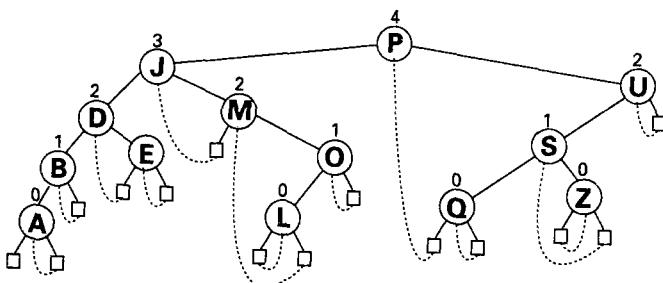


Figura 17.9 Inserción externa en un árbol Patricia.

La Figura 17.9 muestra el resultado de insertar $Z = 11010$ en el árbol Patricia de la Figura 17.8. Como se describió anteriormente, la búsqueda de Z termina en el nodo que contiene a $S = 10011$. Por la propiedad de definición del árbol, S es la única clave del árbol para la que una búsqueda terminaría en ese nodo. Si se inserta Z , habría dos nodos así que el enlace ascendente que se dirigía al nodo que contiene a S debe ponerse ahora a apuntar al nuevo nodo que contiene a Z , con un bit de índice que corresponde al punto más a la izquierda en el que S y Z difieren y con dos enlaces ascendentes: uno apuntando a S y el otro apuntando a Z . Esto corresponde precisamente a reemplazar en la inserción trie por residuos el nodo terminal que contiene a S por un nuevo nodo interno con S y Z como hijos, eliminando una ramificación de una vía por la inclusión del bit de índice.

La inserción de $G = 00111$ ilustra un caso más complicado, como se muestra en la Figura 17.10. La búsqueda de G termina en $E = 00101$, indicando que E es la única clave del árbol con el patrón $001*1$. Ahora, G y E difieren en el bit 1, una posición que se saltó durante la búsqueda. El requisito de que el bit de índice decrezca a medida que se desciende por el árbol exige que G se inserte entre D y E , con un autoapuntador hacia arriba que corresponde a su propio

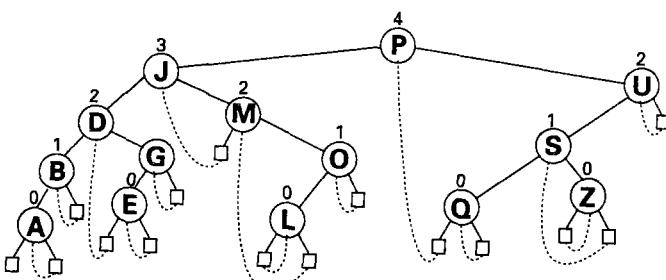


Figura 17.10 Inserción interna en un árbol Patricia.

bit 1. En este árbol destaca el hecho de que el haberse saltado el bit 3 en el subárbol derecho implicase U, S y Q tienen el mismo valor en el bit 3.

Estos ejemplos ilustran los dos únicos casos que se pueden presentar en una inserción en Patricia. La implementación siguiente da los detalles del proceso:

```
void Dicc::insertar(tipoElemento v, tipoInfo info)
{
    struct nodo *p, *t, *x;
    int i = maxb;
    p = cabeza; t = cabeza->izq;
    while (p->b > t->b)
        { p = t; t = (bits(v, t->b, 1)) ? t->der : t->izq; }
    if (v == t->clave) return;
    while (bits(t->clave, i, 1) == bits(v, i, 1)) i--;
    p = cabeza; x = cabeza->izq;
    while (p->b > x->b && x->b > i)
        { p = x; x = (bits(v, x->b, 1)) ? x->der : x->izq; }
    t = new nodo;
    t->clave = v; t->info = info; t->b = i;
    t->izq = (bits(v, t->b, 1)) ? x : t;
    t->der = (bits(v, t->b, 1)) ? t : x;
    if (bits(v, p->b, 1)) p->der = t; else p->izq = t;
}
```

(Este código supone que `cabeza` se inicializa con un campo clave igual a 0, un índice de bit en `maxb` y dos enlaces autoapuntando a `cabeza`.) Primero se hace una búsqueda hasta encontrar la clave que se debe distinguir de `v`. Las condiciones $x->b \leq i$ y $p->b \leq x->b$ caracterizan las situaciones que se muestran en las Figuras 17.10 y 17.9, respectivamente. A continuación se determina la posición del bit más a la izquierda a partir del cual difieren las claves, descendiendo por el árbol hasta ese punto e insertando un nuevo nodo que contenga a `v`.

Patricia es la quintaesencia de los métodos de búsqueda por residuos: permite la identificación de los bits que distinguen las claves de búsqueda y los organiza dentro de una estructura de datos (sin nodos sobrantes) que conduce rápidamente, a partir de cualquier clave de búsqueda, a la única clave de la estructura de datos que pueda ser igual a aquélla. Evidentemente, la misma técnica utilizada en Patricia puede utilizarse en una búsqueda trie por residuos binaria para eliminar las ramificaciones de una sola vía, pero esto no hace más que aumentar el problema de los múltiples tipos de nodos. La Figura 17.11 muestra el árbol Patricia para las mismas claves utilizadas para construir el trie de la Figura 17.6. Este árbol no sólo contiene un 44% de nodos menos, sino que está mejor equilibrado.

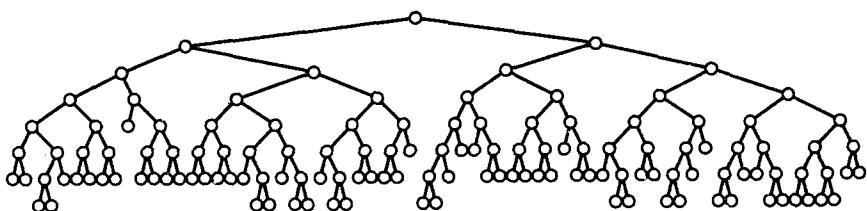


Figura 17.11 Un gran árbol Patricia.

A diferencia de la búsqueda estándar por árbol binario, los métodos por residuos no son sensibles al orden en el que se insertan las claves: dependen solamente de la estructura de las claves en sí mismas. Para Patricia la colocación de los enlaces ascendentes depende del orden de inserción, pero la estructura del árbol depende sólo de los bits de las claves, como en los otros métodos. Así que incluso Patricia podría tener problemas con un conjunto de claves como 001, 0001, 00001, 000001, etc., pero para conjuntos normales de claves, el árbol debe estar relativamente bien equilibrado, por tanto, el número de inspecciones de bits, aun para claves muy grandes, será aproximadamente proporcional a $\lg N$ cuando hay N nodos en el árbol.

Propiedad 17.4 *Un trie Patricia construido a partir de N claves aleatorias de b bits tiene N nodos y necesita $\lg N$ comparaciones de bits para una búsqueda media.*

Al igual que para los otros métodos de este capítulo, el análisis del caso medio es bastante difícil: resulta que Patricia implica una comparación menos, como media, que en el caso de una búsqueda en un trie estándar. ■

La característica más útil de la búsqueda trie por residuos es que se puede realizar eficazmente con claves de longitud variable. En todos los otros métodos de búsqueda que se han visto, la longitud de la clave está de alguna forma «integrada» en el procedimiento de búsqueda, por lo que el tiempo de ejecución depende de la longitud de las claves, así como de su número. Las posibles ganancias que se puedan lograr dependen del método de acceso a los bits que se utilice. Por ejemplo, suponiendo que se dispone de una computadora que puede acceder eficazmente a datos en «octetos» de 8 bits, y que se necesita buscar entre cientos de claves de 1.000 bits, entonces Patricia necesitaría acceder solamente a alrededor de 9 o 10 octetos de la clave de la búsqueda, más una comparación de igualdad de 125 octetos, mientras que la dispersión necesitaría acceder a los 125 octetos de la clave para calcular la función de dispersión, más algunas comparaciones de igualdad, y los métodos basados en comparaciones necesitarían varias comparaciones de gran longitud. Esta propiedad convierte a Patricia (o a la búsqueda por residuos trie sin ramificaciones de una sola vía) en el método de búsqueda a escoger cuando las claves sean muy largas.

Ejercicios

1. Dibujar el árbol de búsqueda digital que se obtiene al insertar las claves C U E S T I O N F A C I L, en este orden, en un árbol inicialmente vacío.
2. Generar un árbol de búsqueda digital de 1.000 nodos y comparar su altura y el número de nodos de cada nivel con los de un árbol de búsqueda binario estándar y con los de un árbol rojinegro (Capítulo 15) construidos sobre las mismas claves.
3. Encontrar un conjunto de 12 claves que generen un árbol de búsqueda digital particularmente mal equilibrado.
4. Dibujar el árbol de búsqueda por residuos que se obtiene al insertar las claves C U E S T I O N F A C I L, en este orden, en un árbol inicialmente vacío.
5. Un inconveniente de una búsqueda por residuos múltiple de 26 vías es que algunas letras del alfabeto se utilizan muy poco. Sugerir una forma de resolver este problema.
6. Describir una forma de suprimir un elemento de un árbol de búsqueda por residuos múltiple.
7. Dibujar el árbol Patricia que se obtiene al insertar las claves C U E S T I O N F A C I L, en este orden, en un árbol inicialmente vacío.
8. Encontrar un conjunto de 12 claves que generen un árbol Patricia particularmente mal equilibrado.
9. Escribir un programa que imprima todas las claves de un árbol Patricia que tengan los mismos t bits iniciales que la clave de búsqueda.
10. ¿Para cuál de los métodos por residuos es razonable escribir un programa que imprima las claves ordenadas? ¿Qué métodos no son aconsejables para esta operación?

Búsqueda externa

Los algoritmos de búsqueda adaptados para acceder a los elementos de archivos muy grandes tienen una inmensa importancia práctica. La búsqueda es la operación fundamental en los grandes archivos de datos, que consume una parte muy significativa de los recursos utilizados en muchos sistemas informáticos.

En este capítulo se centrará el interés sobre todo en los métodos de búsqueda en grandes archivos en disco, puesto que la búsqueda en disco es la de mayor interés práctico. Con dispositivos secuenciales como las cintas, la búsqueda se transforma rápidamente en un método trivial y lento: para buscar un elemento en una cinta no se puede hacer otra cosa que instalarla y leerla hasta encontrar el elemento. Notablemente, los métodos que se estudiarán aquí pueden encontrar un elemento en un disco de una capacidad de hasta un millón de palabras con sólo dos o tres accesos al disco.

Al igual que en la ordenación externa, el aspecto «sistema», unido a la utilización de materiales complejos de E/S, es un factor decisivo en el rendimiento de los métodos de búsqueda externa, pero no será posible estudiarlo con gran detalle. Sin embargo, a diferencia de la ordenación, donde los métodos externos son realmente muy diferentes de los internos, se verá que los métodos de búsqueda externa son prolongaciones lógicas de los métodos que se han estudiado para la búsqueda interna.

La búsqueda es una operación fundamental en los dispositivos de disco. Los archivos se organizan por lo general de forma que se aprovechen las características particulares de los dispositivos para permitir un acceso a la información tan eficaz como sea posible. Como se hizo con la ordenación, se trabajará con un modelo algo simple e impreciso de dispositivos de «discos» con el objeto de exponer las características principales de los métodos fundamentales. La determinación de cuál es el mejor método de búsqueda externa para una aplicación en particular es extremadamente complicada y depende mucho de las características del material (y del software de los sistemas), y por lo tanto está fuera del alcance de este libro. Sin embargo, es posible sugerir algunas concepciones generales a utilizar.

En muchas aplicaciones, con frecuencia se desea poder cambiar, añadir, eliminar o (lo más importante) acceder rápidamente a algunos bits de información de archivos muy grandes. En este capítulo se examinarán algunos métodos para tales situaciones dinámicas, que ofrecen sobre los métodos directos el mismo tipo de ventajas que la búsqueda por árbol binario y la dispersión ofrecen sobre la búsqueda binaria y la secuencial.

Todo gran conjunto de información que se ha de procesar por medio de una computadora se denomina *base de datos*. Se ha realizado gran cantidad de estudios para construir, mantener y utilizar bases de datos. Sin embargo, las grandes bases de datos tienen una inercia muy elevada: una vez que se ha construido una de ellas alrededor de una determinada estrategia de búsqueda resulta muy costoso reconstruirla para otra. Por esta razón, los antiguos métodos estáticos se utilizan ampliamente y quizás se mantengan mucho tiempo, aunque se están comenzando a utilizar nuevos métodos dinámicos en las bases de datos más modernas.

Por lo regular, las aplicaciones de gestión de bases de datos permiten operaciones mucho más complicadas que la simple búsqueda de un elemento por medio de una clave. A menudo las búsquedas se apoyan en criterios que implican más de una clave y que devuelven un gran número de registros. En los últimos capítulos se verán algunos ejemplos de algoritmos adaptados a las peticiones de búsqueda de este tipo, pero las peticiones de búsqueda suelen ser lo suficientemente complejas como para que sea normal hacer una búsqueda secuencial sobre toda la base de datos, evaluando cada registro para ver si satisface los criterios.

Los métodos que se presentan en este capítulo son de importancia práctica en la implementación de sistemas de grandes archivos en los que cada uno tiene un identificador único, con el objeto de permitir el acceso, las inserciones y eliminaciones, basados en dicho identificador. En el modelo que se va a tratar se considerará que el espacio de almacenamiento en disco está dividido en *páginas*, bloques contiguos de información a las que los mecanismos del disco pueden acceder eficazmente. Cada página contendrá muchos registros que se deben organizar dentro de ellas de tal forma que se pueda llegar a cualquier registro leyendo solamente algunas páginas. Se supone que el tiempo de E/S necesario para leer una página domina totalmente al tiempo de procesamiento que se requiere para hacer cualquier cálculo sobre la información que contiene la página. Como se mencionó con anterioridad, este modelo está muy simplificado en ciertos aspectos, pero refleja bastante las características de los dispositivos actuales de almacenamiento externo como para permitir valorar alguno de los métodos fundamentales utilizados.



Figura 18.1 Acceso secuencial.

Acceso secuencial indexado

La búsqueda secuencial en disco es la extensión natural de los métodos de búsqueda secuencial elementales que se estudiaron en el Capítulo 14: los registros se almacenan en orden creciente de sus claves y las búsquedas se efectúan simplemente leyendo los registros uno tras otro hasta encontrar uno que tenga una clave mayor o igual que la buscada. Por ejemplo, si las claves de búsqueda son E J E M P L O D E B U S Q U E D A E X T E R N A y se dispone de discos capaces de contener tres páginas de cuatro registros cada una, entonces se obtiene la configuración que se muestra en la Figura 18.1. (Al igual que para la ordenación en memoria externa, se deben considerar pequeños ejemplos para entender los algoritmos y ejemplos muy grandes para apreciar su rendimiento.) Evidentemente, la búsqueda secuencial pura no es atractiva porque, por ejemplo, buscar W en la Figura 18.1 requeriría leer todas las páginas.

Para mejorar la velocidad de las búsquedas, se puede mantener para cada disco un «índice» que establezca qué claves pertenecen a las páginas de ese disco, como en la Figura 18.2. La primera página de cada disco es su índice: las letras pequeñas indican que sólo se almacena el valor de la clave, no el registro completo, y los números pequeños son los índices de páginas (0 indica la primera página del disco, 1 la siguiente, etc.). En el índice, cada número de página aparece debajo del valor de la última clave de la página anterior. (El espacio en blanco es una clave centinela, menor que todas las otras, y el «+» significa «consultar el disco siguiente».) Así que, por ejemplo, el índice del disco 2 indica que su primera página contiene los registros con claves entre E y J, inclusive, y su segunda página los de claves entre J y O, inclusive. Por lo regular, es posible mantener muchas más claves e índices de páginas en una página de índices que registros en una página de «datos»; de hecho, el índice de un disco completo necesita sólo algunas páginas.

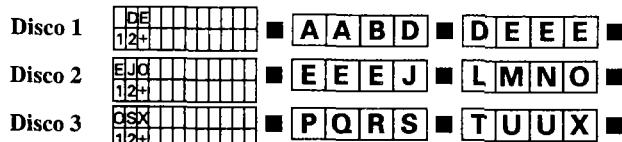


Figura 18.2 Acceso secuencial indexado.

Para acelerar aún más la búsqueda, estos índices pueden estar acoplados con un «índice maestro» que establezca qué claves están en qué discos. En el ejemplo, el índice maestro diría que el disco 1 contiene las claves menores o iguales que E, el disco 2 las claves menores o iguales que O (pero no menores que E) y el disco 3 contiene las claves menores o iguales que X (pero no menores que P). El índice maestro es posiblemente tan pequeño como para tenerlo fijo en memoria, de modo que la mayoría de los registros se pueden encontrar accediendo sólo a dos páginas, una para el índice del disco apropiado y una para la página que contiene el registro apropiado. Por ejemplo, una búsqueda de W implicaría primero la lectura de la página de índices del disco 3 y luego la lectura de la segunda página de datos del disco 3, que es la única que podría contener a W. Las búsquedas de las claves que aparecen en el índice necesitan la lectura de tres páginas: la del índice más las dos páginas que flanquean al valor de la clave del índice. Si no hay claves duplicadas en el archivo se puede evitar el acceso a la página extra. Por otro lado, si hay muchas claves iguales en el archivo, pueden ser necesarios varios accesos a las páginas (registros con claves iguales pueden llenar varias páginas).

Puesto que esto combina una organización secuencial de las claves con un acceso indexado, esta técnica se denomina *acceso secuencial indexado*. Éste es el método a escoger para aplicaciones en las que los cambios en la base de datos son poco frecuentes.

El inconveniente de utilizar el acceso secuencial indexado es que resulta muy rígido. Por ejemplo, para añadir C a la configuración anterior se necesita que la base de datos se reconstruya prácticamente, con nuevas posiciones para la mayor parte de las claves y nuevos valores para los índices.

Propiedad 18.1 *Una búsqueda en un archivo secuencial indexado necesita sólo un número constante de accesos al disco, pero una inserción puede implicar reorganizar el archivo completo.*

De hecho, la «constante» en cuestión depende del número de discos y del tamaño relativo de los registros, los índices y las páginas. Por ejemplo, un gran archivo de claves de una sola palabra no podría estar almacenado en un solo disco de modo que permitiera la búsqueda con un número constante de accesos. O, para tomar otro ejemplo absurdo en el extremo opuesto, un gran número de discos muy pequeños, capaces de contener cada uno un solo registro, harían también difícil la búsqueda.■

Árboles B

Una forma mejor de efectuar la búsqueda en situaciones dinámicas es utilizar árboles equilibrados. Con objeto de reducir el número de los accesos al disco (que son relativamente caros), es razonable permitir un gran número de claves

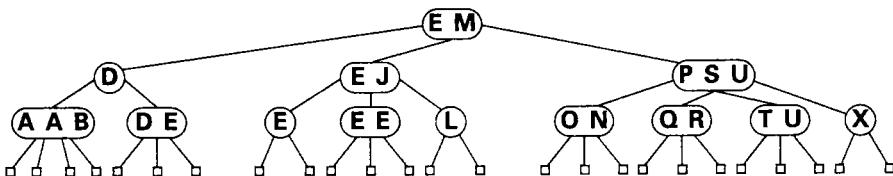


Figura 18.3 Un árbol B.

por nodo, lo que provoca que los nodos tengan un alto grado de ramificación. Tales árboles fueron denominados árboles B por R. Bayer y E. McCreight, que fueron los primeros en considerar el uso de árboles equilibrados múltiples para las búsquedas en memoria externa. (Mucha gente reserva el término árbol B para describir la estructura de datos construida por el algoritmo que sugirieron Bayer y McCreight; aquí se utilizará este nombre como un término genérico que significa «árboles equilibrados externos».)

El algoritmo descendente que se utilizó para los árboles 2-3-4 (ver el Capítulo 15) se generaliza fácilmente para manipular más claves por nodo: supóngase que existe un valor cualquiera entre 1 y $M-1$ de claves por nodo (y por tanto de 2 a M enlaces por nodo). La búsqueda se lleva a cabo en forma análoga a la de los árboles 2-3-4: para desplazarse de un nodo al siguiente, primero se debe encontrar el intervalo apropiado de la clave de búsqueda en el nodo en curso y entonces salir a través del enlace correspondiente hacia el siguiente nodo. Se continúa de esta manera hasta que se alcance un nodo terminal, insertando la nueva clave en el último nodo interno alcanzado. Al igual que en los árboles 2-3-4 descendentes, es necesario «dividir» los nodos que están «llenos» que se encuentran al descender por el árbol: cada vez que se encuentre un k -nodo asociado con un M -nodo, se reemplaza por un $(k+1)$ -nodo asociado a dos $(M/2)$ -nodos (se supone que M es par). Esto garantiza que cuando se alcance el fondo habrá espacio para insertar el nuevo nodo.

El árbol B construido con $M = 4$ para el ejemplo del conjunto de claves se muestra en la Figura 18.3. Este árbol tiene 13 nodos, que corresponden cada uno a una página de disco. Cada nodo puede contener enlaces y registros. El escoger $M = 4$, aun cuando conduce a los familiares árboles 2-3-4, se hace para resaltar este punto: anteriormente se podían fijar cuatro registros por página; ahora sólo se fijarán tres para dejar espacio a los enlaces. La cantidad total de espacio utilizado depende del tamaño relativo de los registros y los enlaces. Posteriormente se verá un método que evita esta mezcla de registros y enlaces.

Al igual que se mantiene en memoria el índice maestro en la búsqueda secuencial indexada, también es razonable mantener en memoria el nodo raíz del árbol B. Para el árbol B de la Figura 18.3, esto permitirá saber que la raíz del subárbol que contiene los registros con claves menores que E están en la página 0 del disco 1, la raíz del subárbol con claves menores que M (pero no menores que E) está en la página 1 del disco 1, y la raíz del subárbol con claves mayores

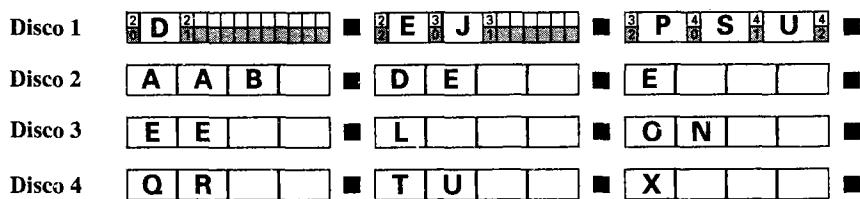


Figura 18.4 Acceso al árbol B.

o iguales que M está en la página 2 del disco 1. Los otros nodos del ejemplo están almacenados como se muestra en la Figura 18.4.

En este ejemplo los nodos se asignan a las páginas del disco recorriendo el árbol de arriba hacia abajo y de derecha a izquierda en cada nivel, asignando nodos al disco 1, luego al disco 2, etc. Se evita almacenar enlaces nulos siguiendo la pista del lugar donde se alcanza el nivel del fondo: en este caso todos los nodos de los discos 2, 3 y 4 tienen todos los enlaces nulos (los cuales no necesitan almacenarse). En una aplicación real entran en juego otras consideraciones. Por ejemplo, pudiera ser mejor evitar que todas las búsquedas tengan que ir a través del disco 1, comenzando la asignación por la página 0 de cada disco, etc. De hecho se necesitan estrategias más sofisticadas debido a la dinámica de la construcción de los árboles (considérese la dificultad de implementar una rutina de *dividir* que respete cualquiera de las estrategias anteriores).

Propiedad 18.2 Una búsqueda o una inserción en un árbol B de orden M con N registros no necesita más de $\log_{M/2}N$ accesos al disco, lo que representa una constante en situaciones prácticas (mientras que M no sea demasiado pequeño).

Esta propiedad se deduce de la observación de que todos los nodos del interior de un árbol B (nodos diferentes de la raíz o las hojas) tienen entre $M/2$ y M claves, puesto que se han formado a partir de una división de un nodo completo con M claves, y cuyo tamaño sólo puede crecer (cuando se divide un nodo inferior). En el peor caso, estos nodos forman un árbol completo de grado $M/2$, que conduce directamente a la cota establecida.■

Propiedad 18.3 Un árbol B de orden M construido a partir de N registros aleatorios contiene aproximadamente $1,44N/M$ nodos.

La demostración de esta afirmación sobrepasa el marco de este libro, pero se puede denotar que la cantidad de espacio perdido llega hasta N , en el peor caso, cuando todos los nodos están medio llenos.■

En el ejemplo anterior ha sido forzosa la elección de $M = 4$ por la necesidad de guardar espacio para los enlaces en los nodos. Pero se acaba *no* utilizando

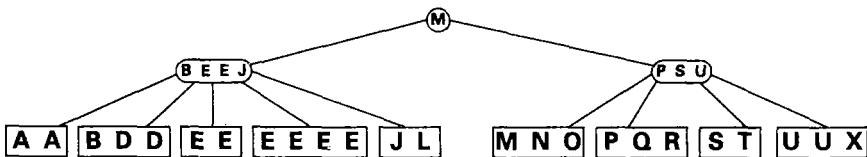


Figura 18.5 Un árbol B con registros sólo en los nodos externos.

enlaces en la mayoría de los nodos, ya que la mayor parte de los nodos de un árbol B son terminales y la mayor parte de los enlaces son nulos. Además, se puede utilizar un valor mucho mayor de M en los niveles más altos del árbol si se almacenan sólo las claves (no los registros completos) en los nodos internos, como en el acceso secuencial indexado. Para comprender cómo sacar partido de estas observaciones en el ejemplo, se supone que se pueden fijar hasta siete claves y ocho enlaces por página, de modo que se puede utilizar $M = 8$ para los nodos internos y $M = 5$ para los nodos del nivel del fondo (*no* $M = 4$ porque en el fondo no se necesita reservar espacio para los enlaces). Un nodo del fondo se divide cuando se le añade un quinto registro (en un nodo con dos registros y un nodo con tres registros); la división termina al «insertar» la clave del registro intermedio en el nodo del nivel superior, donde hay espacio porque el árbol superior ha operado como un árbol B normal con $M = 8$ (sobre las claves almacenadas, no sobre los registros). Esto conduce al árbol que se muestra en la Figura 18.5.

El efecto en una aplicación típica es posiblemente mucho más notorio, puesto que el factor de ramificación del árbol crece aproximadamente en la relación del tamaño del registro con el tamaño de la clave, lo que es susceptible de ser grande. También, con este tipo de organización, el «índice» (que contiene claves y enlaces) puede separarse de los registros reales, como en la búsqueda secuencial indexada. La Figura 18.6 muestra cómo se puede almacenar el árbol de la Figura 18.5: el nodo raíz está en la página 0 del disco 1 (hay espacio para ello, puesto que el árbol de la Figura 18.5 tiene un nodo menos que el árbol de la Figura 18.3), aunque en la mayoría de las aplicaciones probablemente se guarde en memoria, como se hizo antes. Todos los comentarios previos que tie-

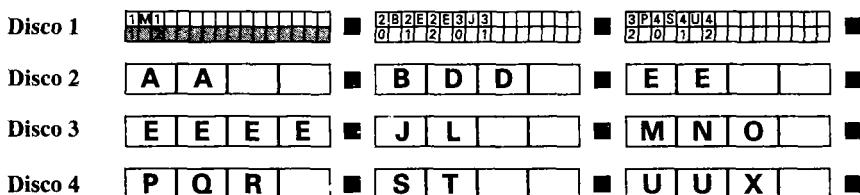


Figura 18.6 Acceso a un árbol B con registros sólo en los nodos externos.

nen que ver con la ubicación de los nodos en los discos son también de aplicación aquí.

Ahora se tienen dos valores de M , uno para los nodos internos, que determina el factor de ramificación del árbol (M_1), y otro para los nodos del fondo del árbol, que determina la asignación de registros a las páginas (M_B). Para minimizar el número de acceso al disco, es preciso hacer que M_1 y M_B sean tan grandes como se pueda, aunque esto suponga cálculos adicionales. Por otra parte, no se desea hacer a M_1 demasiado grande, porque la mayoría de los nodos del árbol estarían muy vacíos y se malgastaría el espacio, y no se desea hacer a M_B demasiado grande, porque esto conduciría a una búsqueda secuencial de los nodos del fondo. Habitualmente, lo mejor es hacer a M_1 y a M_B del tamaño de una página. La elección obvia de M_B es el número de registros que puede tener una página (más uno): el objetivo de la búsqueda es encontrar la página que contiene el registro deseado. Si se toma M_1 como el número de claves que se pueden poner entre dos y cuatro páginas, entonces el árbol B posiblemente tenga sólo tres niveles de profundidad, incluso en archivos muy grandes (un árbol de tres niveles con $M_1 = 2.048$ puede resolver hasta 1.024^3 , o más de mil millones, de entradas.) Pero es preciso recordar que el nodo raíz del árbol, al que se accede para toda operación sobre el árbol, se guarda en memoria, lo que significa que sólo se necesitan dos accesos al disco para encontrar cualquier elemento del archivo.

Como se mencionó brevemente al final del Capítulo 15, con frecuencia se utiliza un método más complicado de inserción «ascendente» para los árboles B (aunque la discusión entre los métodos descendentes y los ascendentes pierde importancia cuando se trata de árboles de tres niveles). En términos técnicos, los árboles descritos aquí deben calificarse como árboles B «descendentes» para distinguirlos de los utilizados comúnmente en la literatura especializada. Se han descrito muchas otras variantes para la búsqueda externa, algunas de ellas muy importantes. Por ejemplo, cuando se llena un nodo, la división (y los nodos semivacíos resultantes) puede anticiparse desplazando una parte de las claves del nodo hacia su nodo «hermano» (si no está demasiado lleno). Esto conduce a una mejor utilización del espacio interior de los nodos, lo que es probablemente uno de los temas más importantes en aplicaciones de búsqueda en disco a gran escala.

Dispersión extensible

Una alternativa a los árboles B, que prolonga los algoritmos de búsqueda digital para aplicarlos en la búsqueda externa, se desarrolló en 1978 por R. Fagin, J. Nievergelt, N. Pippenger y R. Strong. Este método, denominado *dispersión extensible*, implica dos accesos al disco en cada búsqueda en aplicaciones típicas, mientras que al mismo tiempo permite una inserción eficaz. Al igual que en los árboles B, los registros se almacenan en páginas que, cuando se llenan, se divi-

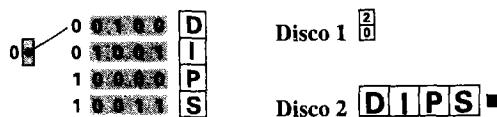


Figura 18.7 Dispersión extensible: primera página.

den en dos partes; como en el acceso secuencial indexado, se mantiene un índice al que se accede para encontrar la página que contiene los registros que concuerdan con la clave de búsqueda. La dispersión extensible combina estas ideas mediante la utilización de las propiedades digitales de las claves de búsqueda.

Para ver cómo funciona la dispersión extensible, considérese la forma en que trata las inserciones sucesivas de las claves del conjunto D I S P E R S I O N E X T E N S I B L E, utilizando páginas con capacidad de hasta cuatro registros. Se comienza con un «índice» con una sola entrada, un puntero a la página que va a contener los registros. Los cuatro primeros registros caben en la página, creando la estructura trivial que se muestra en la Figura 18.7.

El directorio del disco 1 indica que todos los registros están en la página 0 del disco 2, donde se mantienen ordenados por sus claves. Se muestra el valor binario de las claves, utilizando la codificación estándar de cinco bits que consiste en la representación binaria de i con la i -ésima letra del alfabeto. Ahora la página está llena y se debe dividir para poder añadir la clave E = 00101. La estrategia es simple: se ponen los registros cuyas claves comienzan por 0 en una página y aquellos cuyas claves comienzan por 1 en otra. Esto requiere duplicar el tamaño del directorio y colocar parte de las claves de la página 0 del disco 2 en la nueva página, formando la estructura que se muestra en la Figura 18.8.

Ahora se pueden añadir R = 10010, S = 10011 e I = 01001, pero la primera página sigue llena, como se muestra en la Figura 18.9. Se necesita otra división antes de añadir O = 01111, y a continuación se procede de la misma forma que en la primera división, dividiendo la primera página en dos partes, una para las claves que comienzan por 00 y otra para las que comienzan por 01. Lo que no

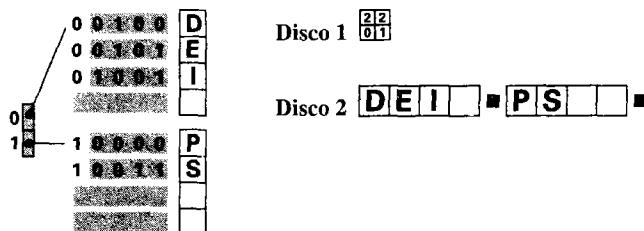


Figura 18.8 Dispersión extensible: división del directorio.

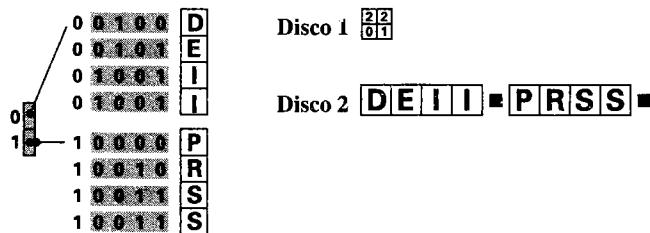


Figura 18.9 Dispersión extensible: primera página otra vez llena.

queda claro de inmediato es qué hacer con el directorio. Una alternativa podría ser simplemente añadir otra entrada, un puntero a cada página. Esto no es muy interesante porque esencialmente conduce a la búsqueda secuencial indexada: el directorio tiene que recorrerse secuencialmente durante cada búsqueda para encontrar la página apropiada. Alternativamente, se puede duplicar otra vez el tamaño del directorio, para obtener la estructura que se muestra en la Figura 18.10. Una nueva página (la página 2 del disco 2) contiene las claves que comienzan por 01 (I o O), la página 0 del disco 2 contiene ahora las claves que comienzan por 00 (X, D y E), y la página que contiene las claves que comienzan por 1 (P, R, y S) no se ha transformado, aunque ahora hay dos punteros dirigidos hacia ella, uno para indicar que las claves que comienzan por 10 están almacenadas allí, el otro para indicar que las claves que comienzan por 11 están almacenadas allí también. Ahora es posible acceder a cualquier registro utilizando los dos primeros bits de su clave para consultar directamente la entrada del directorio que proporciona la dirección de la página que contiene al registro.

Mantener los registros ordenados dentro de la página puede parecer una simplificación por fuerza bruta, pero se recuerda que la hipótesis inicial es que

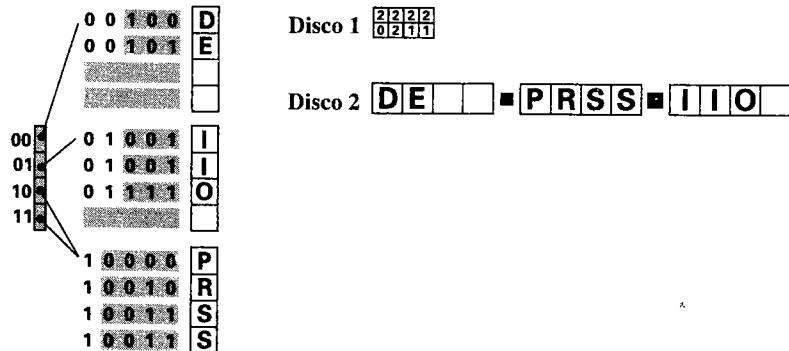


Figura 18.10 Dispersión extensible: segunda división.

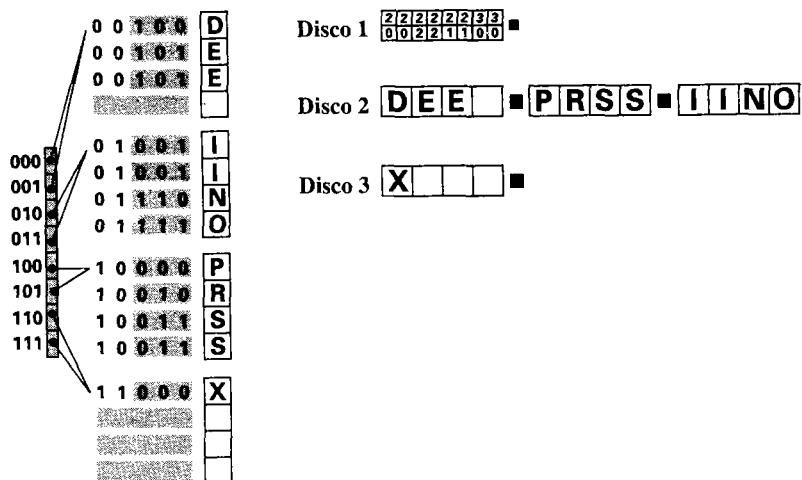


Figura 18.11 Dispersión extensible: tercera división.

se hace la E/S de disco en unidades de páginas y que el tiempo de procesamiento es despreciable comparado con el tiempo de entrada o de salida de una página. De modo que mantener los registros ordenados por sus claves no es un verdadero gasto: para añadir un registro a una página se debe leer la página de la memoria, modificarla y escribirla de nuevo en el disco. El tiempo extra que se necesita para mantener el orden posiblemente no se note en el caso típico en el que las páginas no son muy grandes.

Continuando un poco más con el ejemplo, es necesaria otra división para añadir $X = 11000$. Esta división también requiere duplicar el directorio para producir la estructura que se muestra en la Figura 18.11. El proceso de duplicar el directorio es simple: se lee el directorio antiguo, y después se crea el nuevo escribiendo dos veces cada entrada del antiguo. Esto crea un espacio para el puntero a la nueva página que acaba de crearse por la división.

En general la estructura creada por un dispersión extensible está compuesta por un *directorio* de 2^d palabras (una para cada serie de d bits) y un conjunto de *páginas hojas*, que contienen todos los registros con claves que comienzan por una serie de bits específica (con d bits o menos). Una búsqueda implica utilizar los primeros d bits de la clave como índice dentro del directorio, el cual contiene punteros a las páginas hojas. Después se accede a la página hoja así referenciada y se busca el registro (utilizando cualquier estrategia). Varias entradas de directorio pueden apuntar a la misma página hoja: con mayor precisión, si una página hoja contiene todos los registros cuyas claves comienzan por una serie específica de k bits (los que no están sombreados en las figuras), entonces tendrá 2^{d-k} entradas del directorio apuntando hacia ella. En la Figura 18.11 se

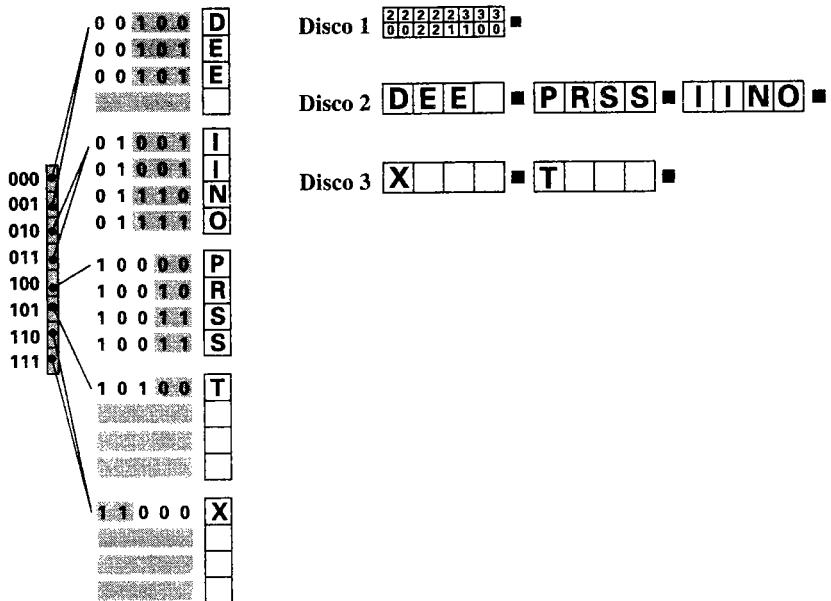


Figura 18.12 Dispersión extensible: cuarta división.

tiene $d = 3$, y la página 1 del disco 2 contiene todos los registros cuyas claves comienzan por los bits 10; por tanto hay dos entradas de directorio apuntando hacia ella.

Hasta ahora, en el ejemplo, cada división de página necesitó una división de directorio, pero en circunstancias normales se puede esperar que el directorio sólo se divida raras veces. Ésta es la esencia del algoritmo: los punteros extra del directorio permiten a la estructura adaptarse armoniosamente a un crecimiento dinámico. Por ejemplo, cuando se inserta T en la estructura de la Figura 18.11, la página 1 del disco 2 se debe dividir para acomodar las cinco claves que comienzan con 10, pero el directorio no necesita crecer, como lo muestra la Figura 18.12. El único cambio en el directorio es que el último de sus dos punteros se cambia para apuntar a la página 1 del disco 3, la nueva página que ha sido creada en la división para acomodar a todas las claves en la estructura de datos que comienzan con 101 (la T).

El directorio sólo contiene punteros a páginas. Éstos son probablemente más pequeños que las claves o los registros; así que cabrán más entradas de directorio en una página. En el ejemplo, se supone que se pueden poner en una página dos veces más entradas a directorios que registros, aunque esta relación posiblemente es mucho más alta en la práctica. Cuando el directorio se expande en más de una página, se mantiene en memoria un «nodo raíz» que indica dónde

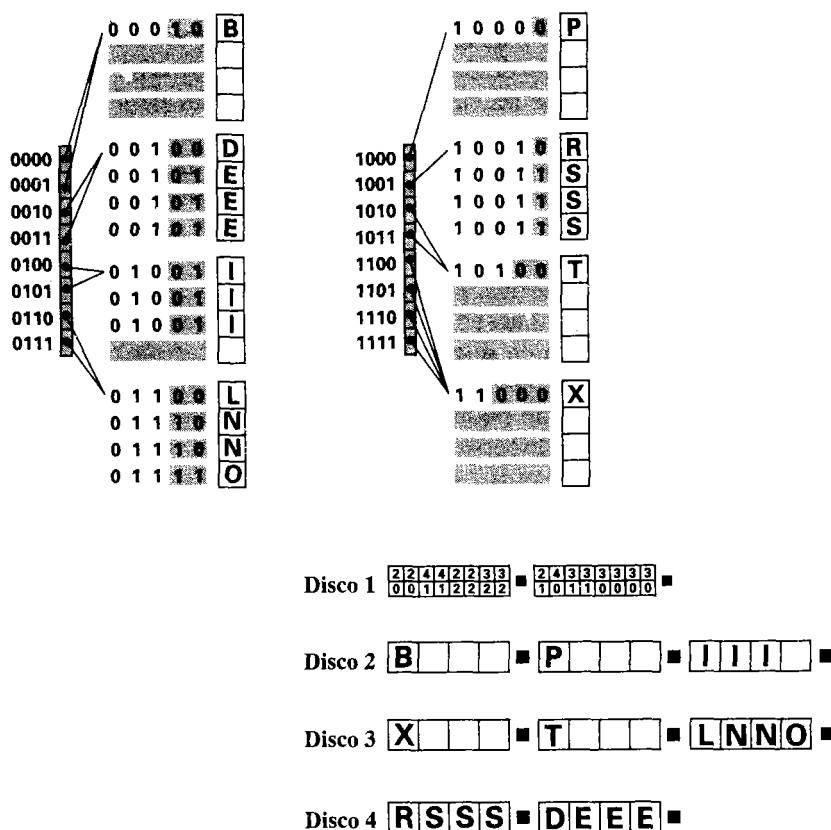


Figura 18.13 Accesos de la dispersión extensible.

están las páginas del directorio, utilizando el mismo esquema de indexación. Por ejemplo, si el directorio se expande en dos páginas, el nodo raíz pudiera indicar que el directorio para todos los registros con claves que comienzan por 0 está en la página 0 del disco 1, y que el directorio para todas las claves que comienzan por 1 está en la página 1 del disco 1. Continuando con el ejemplo, se insertan las claves E N S I B y L, llegando a la estructura que se muestra en la Figura 18.13. (Por claridad, se ha reservado el disco 1 para el directorio, aunque en la práctica pudiera estar mezclado con las otras páginas, o estar reservada la página 0 de cada disco, o bien utilizar alguna otra estrategia.) Así pues, la inserción en una estructura de dispersión extensible puede implicar alguna de las siguientes operaciones, una vez que se acceda a la página hoja que podría contener la clave de búsqueda. Si hay espacio en la página hoja, se inserta simplemente el nuevo registro, o en caso contrario se divide en dos la página hoja (parte de los registros se desplazan hacia una nueva página). Si el directorio tiene más de

una entrada apuntando a la página hoja, entonces las entradas se pueden dividir por igual en la página. Si no, se debe doblar el tamaño del directorio.

Como se ha descrito hasta aquí, este algoritmo es muy sensible a una mala distribución de las claves de entrada: el valor de d es el mayor número de bits que se necesitan para separar las claves en conjuntos lo suficientemente pequeños como para que quepan en las páginas hojas, y así, si un gran número de claves coinciden en gran parte de sus bits iniciales, el directorio se hace inaceptablemente grande. Para aplicaciones reales a gran escala se puede evitar este problema efectuando una dispersión de las claves para hacer los primeros bits (pseudo) aleatorios. Para buscar un registro, se hace una dispersión de su clave para obtener una serie de bits que se utilizarán para acceder al directorio; este último indica en qué página hay que buscar un registro con la misma clave. Desde el punto de vista de la dispersión se puede presentar el algoritmo como una división de nodos para resolver el problemas de las colisiones: de aquí el nombre de «dispersión extensible». Este método ofrece una alternativa atractiva a los árboles B y al acceso secuencial indexado, porque utiliza siempre exactamente dos accesos al disco en cada búsqueda (como el acceso secuencial indexado), mientras que mantiene la capacidad para hacer inserciones eficaces (como los árboles B) sin malgastar mucho espacio.

Propiedad 18.4 *Con páginas que pueden contener M registros, se puede esperar que la dispersión necesite alrededor de $1,44(N/M)$ páginas para un archivo de N registros. El directorio contendrá alrededor de $N^{1+1/M}/M$ entradas.*

Este análisis es una extensión compleja del análisis de los tries a los que se hizo referencia en el capítulo anterior. Cuando M es grande, el volumen de espacio malgastado es aproximadamente el mismo que para los árboles B, pero para un M pequeño el directorio puede hacerse demasiado grande. ■

Aun con la dispersión, se deben dar algunos pasos extra si existe un gran número de claves iguales. Éstas pueden hacer al directorio artificialmente grande, y el algoritmo se distorsiona por completo si hay más claves iguales que las que pueden caber en una página hoja. (Esto ocurre realmente en el ejemplo, puesto que se tienen cuatro E.) Si existen muchas claves iguales entonces se podría, por ejemplo, prohibir la presencia de las mismas en la estructura de datos, y poner en las páginas hojas punteros a listas enlazadas de registros que contengan las claves repetidas. Para ver la complicación que esto implica, considérese lo que pasaría si la última E del ejemplo (que parecía haberse olvidado) se insertara en la estructura de la Figura 18.13.

Una situación menos catastrófica de resolver es que la inserción de una nueva clave pueda causar que el directorio se divida más de una vez. Esto ocurre cuando un bit más no es suficiente para distinguir las claves en una página sobrecargada. Por ejemplo, si se insertaran dos claves con el valor D = 00100 en la estructura de dispersión extensible de la Figura 18.12, se necesitarían dos divisiones del directorio porque se necesitan cinco bits para distinguir D de E (el

cuarto bit no ayuda). Esto es fácil de afrontar en una implementación, pero no se debe tolerar.

Memoria virtual

El «método más fácil» que se presentó al final del Capítulo 13 para la ordenación externa se puede aplicar directa y trivialmente al problema de la búsqueda externa. En realidad, una memoria virtual no es más que un método de búsqueda externa de amplio espectro: dada una dirección (clave), devolver la información asociada con esa dirección. Sin embargo, la utilización directa de la memoria virtual *no* se recomienda como método fácil de búsqueda. Como se mencionó en el Capítulo 13, las memorias virtuales se comportan mejor cuando la mayoría de los accesos están relativamente próximos a los accesos anteriores. Los algoritmos de ordenación se pueden adaptar a esto, pero la verdadera naturaleza de la búsqueda es que las peticiones traten sobre informaciones de las partes arbitrarias de la base de datos.■

Ejercicios

1. Dar el contenido del árbol B que se obtiene de la inserción de las claves C U E S T I O N F A C I L en un árbol inicialmente vacío y con $M = 5$.
2. Dar el contenido del árbol B que se obtiene de la inserción de las claves C U E S T I O N F A C I L en un árbol inicialmente vacío y con $M = 6$. Utilizar la variante del método en el que todos los registros se conserven en nodos externos.
3. Dibujar el árbol B que se obtiene al insertar dieciséis claves iguales en un árbol inicialmente vacío, con $M = 5$.
4. Suponiendo que se destruye una página de una base de datos, describir cómo se resolvería este problema para cada una de las estructuras de árboles B descritas en el texto.
5. Dar el contenido de la tabla de dispersión extensible que se obtiene cuando se insertan las claves C U E S T I O N F A C I L en una tabla inicialmente vacía, con capacidad de página para cuatro registros. (Siguiendo el ejemplo del texto, no se debe utilizar la dispersión sino la representación binaria de 5 bits de i como clave para la i -ésima letra.)
6. Obtener una secuencia de tantas claves distintas como sea posible que hagan crecer un directorio desde una tabla inicialmente vacía hasta un tamaño 16, con una capacidad de página de tres registros.
7. Esbozar un método para *suprimir* un elemento de una tabla de dispersión extensible.

8. ¿Por qué los árboles B «descendentes» son mejores que los «ascendentes» para el acceso concurrente a los datos? (Supóngase, por ejemplo, que dos programas están tratando de insertar un nuevo nodo al mismo tiempo.)
9. Implementar *buscar* e *insertar* para una búsqueda *internal* utilizando el método de dispersión extensible.
10. Comparar el programa del ejercicio anterior con la doble dispersión y la búsqueda *trie* por residuos, en aplicaciones de búsqueda interna.

REFERENCIAS para la Búsqueda

Las referencias principales para esta sección son el Volumen 3 de Knuth, el libro de Gonnet y el libro de Mehlhorn. La mayoría de los algoritmos que se han estudiado se tratan detalladamente en estos libros, con análisis matemáticos y sugerencias para aplicaciones prácticas. Los métodos clásicos son tratados por Knuth y los más recientes por Gonnet y Mehlhorn, con muchas referencias bibliográficas. Estas tres fuentes describen los análisis de casi todos los «fuera del alcance de este libro» a los que se ha hecho referencia en esta sección.

El material del Capítulo 15 proviene del artículo de 1978 de Guibas y Sedgewick, que muestra cómo adaptar muchos algoritmos clásicos de árboles equilibrados al esquema «rojinegro», a la vez que ofrece otras implementaciones. En realidad, hay una literatura muy amplia sobre árboles equilibrados: el lector que desee ampliar sus conocimientos puede comenzar con este artículo. El libro de Mehlhorn da pruebas detalladas de las propiedades de los árboles rojinegros y de estructuras similares, así como referencias a trabajos más recientes. El estudio realizado por Comer en 1979 presenta los árboles B desde un punto de vista más práctico.

El algoritmo de la dispersión extensible que se presentó en el Capítulo 18 proviene del artículo de Fagin, Nievergelt, Pippenger y Strong de 1979. Este artículo es obligatorio para todo aquel que desee más información sobre los métodos de búsqueda externa: relaciona el contenido de los Capítulos 16 y 17 hasta ofrecer el algoritmo del Capítulo 18. El artículo contiene también un análisis detallado y una presentación de las consecuencias prácticas.

Muchas aplicaciones prácticas de los métodos expuestos, especialmente en el Capítulo 18, provienen del contexto de los sistemas de bases de datos. El estudio de las bases de datos es un campo amplio y en crecimiento, en el que los algoritmos básicos de búsqueda continúan desempeñando un papel fundamental en la mayoría de los sistemas. El libro de Ullman es una introducción a este campo.

- D. Comer, «The ubiquitous B-tree», *Computing Surveys*, 11 (1979).
- R. Fagin, J. Nievergelt, N. Pippenger y H. R. Strong, «Extendible dispersion—a fast access method for dynamic files», *ACM Transactions on Database Systems*, 4, 3 (septiembre 1979).
- G. H. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1984.
- L. Guibas y R. Sedgewick, «A dichromatic framework for balanced trees», en *19th Annual Symposium on Foundations of Computer Science*, IEEE, 1978. También en *A Decade of Progress 1970-1980*, Xerox PARC, Palo Alto, CA.
- D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlín, 1984.
- J. D. Ullman, *Principle of Database Systems*, Computer Science Press, Rockville, MD, 1982.

Procesamiento de cadenas

Búsqueda de cadenas

A menudo sucede que los datos a procesar no se descomponen lógicamente en registros independientes que representen pequeñas partes identificables. Este tipo de datos se caracteriza fácilmente por el hecho de que se pueden escribir en forma de *cadenas*: series lineales (por lo regular muy largas) de caracteres. Por supuesto que ya se han visto antes las cadenas, por ejemplo en los Capítulos 3 y 16, ya que constituyen estructuras básicas en C++.

Las cadenas son evidentemente el centro de los sistemas de tratamiento de texto, que proporcionan una gran variedad de posibilidades para la manipulación de textos. Tales sistemas procesan *cadenas alfanuméricas*, que pueden definirse en primera aproximación como series de letras, números y caracteres especiales. Estos objetos pueden ser bastante grandes (por ejemplo, este libro contiene más de un millón de caracteres), por lo que es importante disponer de algoritmos eficaces para su manipulación.

Otro tipo de cadena es la *cadena binaria*, que es una simple serie de valores 0 y 1. Ésta es, en cierto sentido, un tipo especial de cadena alfanumérica, pero es útil hacer la distinción porque existen diferentes algoritmos específicos para este tipo de cadenas y porque las cadenas binarias se utilizan en muchas aplicaciones. Por ejemplo, algunos sistemas gráficos de computadoras representan las imágenes como cadenas binarias. (Este libro fue impreso con un sistema de este tipo: esta misma página se representó en su momento como una cadena binaria de millones de bits.)

En un sentido, las cadenas alfanuméricas son objetos bastante diferentes de las cadenas binarias, porque están constituidas por caracteres tomados de un gran alfabeto. Pero, en otro sentido, los dos tipos de cadenas son equivalentes, puesto que cada carácter alfanumérico se puede representar (por ejemplo) con ocho cifras binarias, y una cadena binaria puede considerarse como una alfanumérica, al tratar cada paquete de ocho bits como un carácter. Se verá que el tamaño del alfabeto que se toma para formar una cadena es un factor importante en el diseño de los algoritmos de procesamiento de cadenas.

Una operación fundamental sobre las cadenas es el *reconocimiento de patro-*

nes: dada una *cadena alfanumérica* de longitud N y un *patrón* de longitud M , encontrar una ocurrencia del patrón dentro del texto. (Se utiliza aquí el término «texto» aun cuando se haga referencia a una secuencia de valores 0 y 1 o a algún otro tipo especial de cadena.) La mayoría de los algoritmos para este problema se pueden modificar fácilmente para encontrar *todas* las ocurrencias del patrón en el texto, puesto que recorren el texto en secuencia y se pueden reiniciar en la posición situada inmediatamente después del comienzo de una concordancia, para encontrar la concordancia siguiente.

El problema del reconocimiento de patrones se puede caracterizar como un problema de búsqueda en el que el patrón sería la clave, pero los algoritmos de búsqueda que se han estudiado no se pueden aplicar directamente porque el patrón puede ser largo y porque se «alinea» en el texto de forma desconocida. Éste es un problema interesante: hace poco tiempo que se ha descubierto que algunos algoritmos muy diferentes (y sorprendentes) no solamente ofrecen un abanico de métodos prácticos, sino que también ilustran algunas de las técnicas fundamentales del diseño de algoritmos.

Una breve historia

Los algoritmos que se van a estudiar tienen una historia interesante que se resume aquí para ayudar a situar a los diferentes métodos en su contexto.

Existe un algoritmo evidente de fuerza bruta para el procesamiento de cadenas que se utiliza ampliamente. Mientras que en el peor caso se ejecuta en un tiempo proporcional a MN , las cadenas con las que se trabaja en muchas aplicaciones conducen a un tiempo de ejecución que es virtualmente proporcional a $M + N$. Además, como el método se puede beneficiar de los recursos de las arquitecturas de la mayoría de los sistemas de computadoras, la versión optimizada del algoritmo constituye un «estándar» que es difícil de batir por un algoritmo más fino.

En 1970, S.A. Cook demostró un resultado teórico sobre un tipo particular de máquina abstracta que implicaba la existencia de un algoritmo para resolver el problema del reconocimiento de patrones en un tiempo proporcional a $M + N$ en el peor caso. D. E. Knuth y V. R. Pratt siguieron laboriosamente el razonamiento que Cook había hecho para probar su teorema (cuya intención no era la de ser práctico) y obtuvieron un algoritmo que pudieron afinar en un método relativamente simple y práctico. Esto es un ejemplo raro y satisfactorio de un resultado teórico con una solución práctica inmediata (e inesperada). Pero resulta que J.H. Morris descubrió prácticamente el mismo algoritmo como solución de un molesto problema práctico que había encontrado cuando implementaba un editor de texto (no deseaba «retroceder» nunca en la cadena de texto). Sin embargo, el hecho de que el mismo algoritmo surgiera de dos aproximaciones diferentes aumenta su credibilidad como una solución fundamental al problema.

Knuth, Morris y Pratt no publicaron su algoritmo hasta 1976, y mientras tanto R. S. Boyer y J. S. Moore (e independientemente, R. W. Gosper) habían descubierto un algoritmo mucho más rápido en muchas aplicaciones, puesto que sólo examina una parte reducida de los caracteres de la cadena de texto. Muchos editores de texto utilizan este algoritmo para obtener una reducción notable del tiempo de respuesta en las búsquedas de cadenas.

Tanto el algoritmo de Knuth-Morris-Pratt como el de Boyer-Moore requieren cierto procesamiento algo complicado del patrón, que es difícil de entender, por lo que se ha limitado su utilización. (De hecho, la historia dice que un programador desconocido encontró el algoritmo de Morris tan difícil de entender que lo reemplazó por una implementación del algoritmo de fuerza bruta.)

En 1980, R. M. Karp y M. O. Rabin observaron que el problema no es tan diferente como parece del problema de una búsqueda estándar, y obtuvieron un algoritmo casi tan simple como el de la fuerza bruta, que realmente siempre se ejecuta en un tiempo proporcional a $M + N$. Además, su algoritmo se adapta fácilmente a patrones y textos bidimensionales, lo que lo hace más útil que otros para el procesamiento de imágenes.

Esta historia ilustra el hecho de que la búsqueda de un «algoritmo mejor» muy a menudo es justificada; incluso se puede pensar que existen aún otras soluciones para el desarrollo de este problema.

Algoritmo de fuerza bruta

El método en el que se piensa de inmediato para el reconocimiento de patrones consiste simplemente en verificar, para cada posición posible del texto en la que el patrón puede concordar, si efectivamente lo hace. El programa siguiente efectúa de esta manera una búsqueda de la primera ocurrencia de la cadena patrón *p* en una cadena texto *a*:

```
int busquedabruna(char *p, char *a)
{
    int i, j, M = strlen(p), N = strlen(a);
    for (i = 0, j = 0; j < M && i < N; i++, j++)
        if (a[i] != p[j]) { i = j-1; j = -1; }
    if (j == M) return i-M; else return i;
}
```

El programa conserva un puntero (*i*) en el texto y otro (*j*) en el patrón. Mientras que apunten a caracteres que concuerden, ambos se incrementan. Si *i* y *j* apuntan a caracteres incompatibles, entonces *j* se pone de nuevo a apuntar al principio del patrón e *i* se reinicializa de forma que se haga avanzar al patrón a la siguiente posición a la derecha, para una nueva comparación. En particular,

Figura 19.1 Búsqueda por fuerza bruta de una cadena en un texto binario.

siempre que la sentencia `if` ponga `j` a -1 , las iteraciones siguientes del bucle `for` van incrementando `i` hasta que se encuentre un carácter del texto que concuerde con el primer carácter del patrón.

Si se alcanza el final del patrón ($j == M$) entonces hubo concordancia a partir de $a[i-M:j]$. Si por el contrario se alcanza el final del texto ($i == N$) antes que el del patrón, entonces no hay concordancia: el patrón no aparece en el texto, en cuyo caso se devuelve el valor «centinela» N .

En una aplicación de edición de texto, el bucle interno de este programa rara vez se reitera, y el tiempo de ejecución es prácticamente proporcional al número de caracteres examinados en el texto. Por ejemplo, suponiendo que se está buscando el patrón DENAS en la cadena de texto

EJEMPLO DE BÚSQUEDA DE CADENAS

entonces la sentencia `j++` se ejecuta sólo cinco veces (dos para cada DE y una para la D de DA) antes de que se encuentre la verdadera concordancia.

Por otra parte, la fuerza bruta puede ser muy lenta para algunos patrones, por ejemplo, si el texto es binario (dos caracteres), como sucede en aplicaciones de procesamiento de imágenes y de programación de sistemas. La Figura 19.1 muestra lo que sucede cuando se utiliza este algoritmo para buscar el patrón 10100111 en una gran cadena binaria. Cada línea (excepto la última, que muestra la concordancia) contiene la lista de los cero o más caracteres que concuerdan con el patrón, seguida de una discordancia. Estas líneas son los «falsos principios» que ocurren cuando se trata de encontrar el patrón; un objetivo evidente del diseño de un algoritmo es tratar de limitar el número y la longitud de dichas líneas. En este ejemplo se examinan por término medio dos caracteres por cada posición de texto, aunque la situación puede ser mucho peor.

Propiedad 19.1 *La búsqueda de cadenas por fuerza bruta puede necesitar alrededor de NM comparaciones de caracteres.*

El peor caso se produce cuando patrón y texto están formados por uno o varios 0 seguidos por un 1. Entonces para cada una de las $N - M + 1$ posiciones de concordancia se comparan con el texto todos los caracteres del patrón, con un coste total de $M(N - M + 1)$. Normalmente M es muy pequeño comparado con N , por lo que el total es alrededor de NM . ■

Por supuesto que tales cadenas degeneradas son poco probables en un texto normal (o en C++), pero pueden aparecer cuando se procesan textos binarios, de modo que hay que buscar mejores algoritmos.

Algoritmo de Knuth-Morris-Pratt

La idea básica de este algoritmo descubierto por Knuth, Morris y Pratt es la siguiente: cuando se detecta una discordancia (no concordancia), el «falso principio» se compone de los caracteres que se conocen por adelantado (puesto que están en el patrón). De algún modo hay que ser capaces de aprovecharse de esta información en lugar de retroceder el puntero i más allá de todos estos caracteres conocidos.

Como un ejemplo sencillo de esto, se supone que el primer carácter del patrón sólo aparece una vez (sea, por ejemplo, el patrón = 10000000). Supóngase ahora que se tiene un falso principio de j caracteres de longitud en alguna posición del texto. Cuando se detecta la no concordancia, se sabe, en virtud del hecho de que concuerdan j caracteres, que no se necesita «retroceder» el puntero i del texto, puesto que ninguno de los $j - 1$ caracteres del texto pueden concordar con el primer carácter del patrón. Este cambio se podría implementar reemplazando la instrucción $i = j - i$ del programa anterior por $i++$. El efecto práctico de este ejemplo es limitado, porque es poco probable que se presenten patrones tan específicos, pero merece la pena pensar en esta idea, y el algoritmo de Knuth-Morris-Pratt es una generalización de la misma. Sorprendentemente siempre es posible arreglar las cosas de modo tal que el puntero i nunca se decremente.

Saltar todos los caracteres del patrón cuando se detecta una discordancia, como la descrita en el párrafo anterior, sería un error en el caso en el que el patrón se repita en el propio punto de la no concordancia. Por ejemplo, cuando se está buscando 10100111 en 1010100111, se comienza por detectar la discordancia en el quinto carácter, pero se debe retroceder al tercero para continuar la búsqueda, puesto que de lo contrario se perdería la concordancia. En todo caso se puede prever la acción a tomar, adelantándose en el tiempo, porque depende sólo del patrón, como se muestra en la Figura 19.2.

Se utilizará el array $\text{prox}[M]$ para determinar cuánto se debe retroceder

j	prox [j]	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
1	0	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
2	0	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
3	1	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
4	2	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
5	0	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
6	1	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1
7	1	1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1

Figura 19.2 Posiciones de reinicialización en una búsqueda de Knuth-Morris-Pratt.

cuando se detecte que no hay concordancia. Imagínese que se hace deslizar una copia de los primeros j caracteres del patrón, de izquierda a derecha, comenzando por colocar el primer carácter de la copia sobre el segundo carácter del patrón y parando cuando todos los caracteres que se superpongan concuerden (o no haya ninguno). Estos caracteres que se superponen definen la siguiente posición posible en la que el patrón podría concordar, si se detecta que no hay concordancia en $p[j]$. La distancia a retroceder en el patrón ($\text{prox}[j]$) es exactamente el número de caracteres que se superponen. Específicamente, para $j > 0$, el valor de $\text{prox}[j]$ es el mayor valor de $k < j$ para el que los primeros k caracteres del patrón concuerdan con los últimos k caracteres de los j primeros caracteres del patrón. Como pronto se verá, es conveniente definir $\text{prox}[0]$ como -1 .

Este array prox proporciona de inmediato una forma de limitar (y de hecho, como se verá, de eliminar) el «retroceso» del puntero i del texto, como se presentó anteriormente. Cuando i y j apuntan a caracteres que no concuerdan (la comprobación de la concordancia comenzó en la posición $i - j + 1$ dentro de la cadena de texto), entonces la próxima posición posible para que haya una concordancia con el patrón es $i - \text{prox}[j]$. Pero por definición de la tabla prox , los primeros $\text{prox}[j]$ caracteres después de esa posición concuerdan con los primeros $\text{prox}[j]$ caracteres del patrón, por lo tanto no hay necesidad de hacer retroceder al puntero i tan lejos: simplemente se puede dejar al puntero i sin cambios y darle al puntero j el valor $\text{prox}[j]$, como se hace en el programa siguiente:

```
int busquedaKMP(char *p, char *a)
{
    int i, j, M = strlen(p), N = strlen(a);
```

```

inicprox(p);
for (i = 0, j = 0; j < M && i < N; i++, j++)
    while ((j >= 0) && (a[i] != p[j])) j = prox[j];
    if (j == M) return i-M; else return i;
}

```

Cuando $j = 0$ y $a[i]$ no concuerdan con $p[0]$, no hay superposición, por lo que se desea incrementar i y mantener j apuntando al comienzo del patrón. Esto se logra definiendo $prox[0]$ en -1 , lo que provoca que a j se le asigne -1 en el bucle while; entonces se incrementa i y j se pone a 0 cuando se itera el bucle for. Funcionalmente este programa es el mismo que el de búsqueda-bruta, pero es probable que se ejecute con más rapidez en patrones que sean altamente repetitivos.

Queda por calcular la tabla prox. El programa correspondiente necesita algo más de astucia; básicamente es el mismo anterior, pero haciendo concordar al patrón consigo mismo.

```

inicprox(char *p)
{
    int i, j, M = strlen(p);
    prox[0] = -1;
    for (i = 0, j = -1; i < M; i++, j++, prox[i] = j)
        while ((j >= 0) && (p[i] != p[j])) j = prox[j];
}

```

Justo después de que se hayan incrementado i y j , se ha determinado que los j primeros caracteres del patrón concuerden con los caracteres de las posiciones $p[i-j-1], \dots, p[i-1]$, los últimos j caracteres de los i primeros caracteres del patrón. Y éste es el mayor j con esta propiedad, puesto que, si no, se habría olvidado una «possible concordancia» del patrón consigo mismo. Así que j es exactamente el valor que se debe asignar a $prox[j]$.

Una forma interesante de representar este algoritmo es considerar al patrón como si estuviera fijo, de forma que la tabla prox pueda «volcarse en» el programa. Por ejemplo, el programa siguiente es exactamente equivalente al programa anterior para el patrón que se está considerando, pero es posible que sea mucho más eficaz.

```

int busquedaKMP(char *a)
{
    int i = -1;
    sm: i++;
    s0: if (a[i] != '1') goto sm; i++;
    s1: if (a[i] != '0') goto s0; i++;
}

```

```

s2: if (a[i] != '1') goto s0; i++;
s3: if (a[i] != '0') goto s1; i++;
s4: if (a[i] != '0') goto s2; i++;
s5: if (a[i] != '1') goto s0; i++;
s6: if (a[i] != '1') goto s1; i++;
s7: if (a[i] != '1') goto s1; i++;
    return i-8;
}

```

Las etiquetas goto corresponden precisamente a la tabla prox. De hecho, el programa inicprox anterior que construye la tabla prox se puede modificar fácilmente para *dar como salida este programa!* Para evitar tener que verificar si $i == N$ cada vez que se incrementa i , se supone que el patrón se almacena al final del texto como un centinela, es decir en $a[N], \dots, a[N+M-1]$. (Esta mejora se puede hacer incluso en la implementación estándar.) Éste es un ejemplo de un «compilador de búsqueda de cadenas»: dado un patrón, se puede generar un programa muy eficaz para buscar ese patrón en una cadena texto arbitrariamente larga. Se verá la generalización de este concepto en los dos capítulos que siguen.

El programa anterior utiliza solamente algunas operaciones muy básicas para resolver el problema de la búsqueda de cadenas. Esto significa que se puede describir en términos de un modelo muy simple de máquina denominada *máquina de estados finitos*. La Figura 19.3 muestra la máquina de estados finitos para el problema anterior.

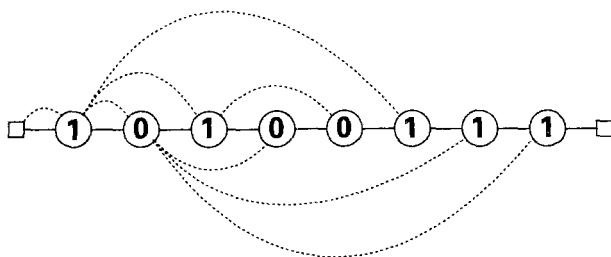


Figura 19.3 Máquina de estados finitos para el algoritmo de Knuth-Morris-Pratt.

La máquina consiste en *estados* (indicados por los números encerrados en círculos) y *transiciones* (indicadas por líneas). Cada estado tiene dos transiciones que salen de él: una transición de *concordancia* (expresada en la figura por las líneas gruesas que van hacia la derecha) y una transición de *no concordancia* (expresada por las líneas de puntos que van hacia la izquierda). Los estados son los lugares donde la máquina ejecuta las instrucciones; las transiciones son las instrucciones goto. Cuando la máquina está en el estado etiquetado « x » puede llevar a cabo una sola instrucción: «si el carácter en curso es x pasa al siguiente

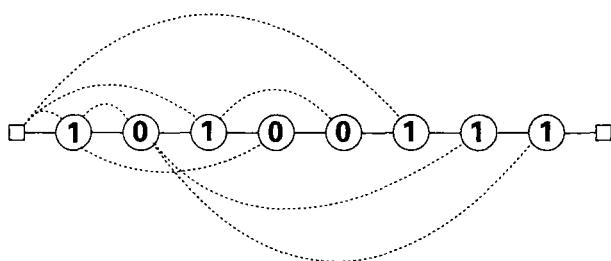


Figura 19.4 Máquina de estados finitos (mejorada) para el algoritmo de Knuth-Morris-Pratt.

y toma la transición de concordancia; en caso contrario toma la transición de «no concordancia». Pasar al siguiente significa tomar el próximo carácter de la cadena como «carácter actual»; la máquina va pasando al carácter siguiente a medida que va concordando con el carácter actual. Hay dos excepciones a esto: el primer estado siempre toma una transición de concordancia y pasa al siguiente carácter (esencialmente esto corresponde a buscar la primera ocurrencia del primer carácter del patrón), y el último estado es un estado de «parada» que indica que se ha encontrado una concordancia del patrón. En el próximo capítulo se verá cómo utilizar una máquina similar (pero más poderosa) para ayudar al desarrollo de un algoritmo mucho más eficaz de reconocimiento de patrones.

El lector atento puede haber notado que es posible mejorar este algoritmo, porque no tiene en cuenta al carácter que causa que no haya concordancia. Por ejemplo, supóngase que el texto comienza por 1011 y que se está buscando el patrón 10100111. Después de la concordancia de 101 se encuentra una no concordancia en el cuarto carácter; en este punto la tabla prox dice que hay que comparar el segundo carácter del patrón con el cuarto carácter del texto, puesto que, en virtud de la concordancia de 101, el primer carácter del patrón se puede alinear con el tercer carácter del texto (pero no hay que compararlos, ya que se sabe que los dos son 1). Sin embargo, sería imposible tener aquí una concordancia: al constatar esta no concordancia se sabe que el próximo carácter del texto no es 0, como lo exige el patrón. Otra forma de ver las cosas es observar la versión del programa que tiene dentro la tabla prox: en la etiqueta 4 se va a 2 si $a[i]$ no es 0, pero en la etiqueta 2 se va a 1 si $a[i]$ no es 0. ¿Por qué no ir directamente a 1? La Figura 19.4 muestra la versión mejorada de la máquina de estados finitos para el ejemplo.

Afortunadamente, es fácil introducir este cambio en el algoritmo. Sólo se necesita reemplazar la sentencia $\text{prox}[i] = j$ en el programa `inicprox` por

$$\text{prox}[i] = (\text{p}[i] == \text{p}[j]) ? \text{prox}[j] : j$$

Puesto que se procede de izquierda a derecha, el valor que se necesita para `prox` ya ha sido calculado, por lo que simplemente se utiliza.

```

1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1
1 0 1 0 0 1 1 1
1 0 1 0 0 1 1 1
1 0 1 0 0 1 1 1
1 0 1 0 0 1 1 1
1 0 1 0 0 1 1 1
1 0 1 0 0 1 1 1
1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1

```

Figura 19.5 Búsqueda de cadenas Knuth-Morris-Pratt en un texto binario.

Propiedad 19.2 *La búsqueda de cadenas Knuth-Morris-Pratt nunca efectúa más de $N+M$ comparaciones de caracteres.*

Esta propiedad se ilustra en la Figura 19.5, y también se deduce del código: o se incrementa j o se reinicializa a partir de la tabla prox, una vez para cada i . ■

La Figura 19.5 muestra que este método verdaderamente efectúa menos comparaciones que el de la fuerza bruta para el ejemplo binario. Sin embargo, es posible que el algoritmo de Knuth-Morris-Pratt no sea mucho más rápido que el método de la fuerza bruta en muchas aplicaciones reales, porque de hecho pocas aplicaciones implican búsquedas de patrones altamente repetitivos en textos también altamente repetitivos. Sin embargo, este método tiene una ventaja práctica fundamental: procede secuencialmente a través del texto de entrada y nunca retrocede. Esto lo hace conveniente para su utilización en grandes archivos que se estén leyendo de algún dispositivo externo. (En estos casos los algoritmos que necesiten retroceder se acompañan de algún complejo sistema de almacenamiento intermedio.)

Algoritmo de Boyer-Moore

Si no es difícil «retroceder» se puede desarrollar un método de búsqueda de cadenas bastante más rápido recorriendo el patrón de derecha a izquierda mientras se está tratando de hacer concordar éste con el texto. Si al buscar el patrón de ejemplo 10100111, se encuentra concordancia en los caracteres octavo, séptimo y sexto, pero no el quinto, se puede deslizar el patrón siete posiciones a la derecha, y volver a comprobar a partir del carácter decimoquinto, porque la concordancia parcial encontró 111, que podría aparecer en cualquier parte del patrón. Por supuesto que al final el patrón suele aparecer en cualquier lugar y, por tanto, se necesita una tabla prox como la anterior.

La Figura 19.6 muestra una versión de derecha a izquierda de la tabla prox para el patrón 10110101: en este caso prox[j] es el número de posiciones de caracteres que se puede desplazar el patrón hacia la derecha, dado que en una

j	prox [j]	
2	4	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1
3	7	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1
4	2	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1
5	5	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1
6	5	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1
7	5	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1
8	5	1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1

Figura 19.6 Posiciones de reinicialización para la búsqueda de Boyer-Moore.

exploración de derecha a izquierda se constató una no concordancia en el j -ésimo carácter desde la derecha del patrón. Este valor se encuentra igual que antes, deslizando una copia del patrón sobre los últimos j caracteres del mismo, de izquierda a derecha, comenzando por alinear el penúltimo carácter de la copia con el último carácter del patrón y parándose cuando todos los caracteres que se superponen concuerden (teniendo en cuenta también el carácter que provocó el fallo de la concordancia). Por ejemplo, $\text{prox}[2]$ es 7 porque, si hay una concordancia de los dos últimos caracteres y luego una no concordancia en una exploración de derecha a izquierda, entonces se debe haber encontrado 001 en el texto; esta serie no aparece en el patrón, excepto posiblemente si el 1 se alinea con el primer carácter del patrón, por lo que se puede deslizarlo 7 posiciones a la derecha.

Esto lleva directamente a un programa que es muy similar a la implementación anterior del método de Knuth-Morris-Pratt. No se estudiará con más detalle porque existe un método completamente diferente de saltar caracteres en una exploración del patrón de izquierda a derecha, que es mejor en muchos casos.

La idea es decidir lo que se debe hacer en función del carácter que provocó la discordancia tanto en el *texto* como en el patrón. La etapa del preprocesamiento consiste en decidir, para cada carácter que podría figurar en el texto, lo que se haría si dicho carácter hubiese provocado la no concordancia. La realización más simple de esto conduce inmediatamente a un programa bastante útil.

La Figura 19.7 muestra este método para el primer ejemplo de texto. Procediendo de derecha a izquierda para hacer concordar al patrón, se comprueba primero la S del patrón con la P (el quinto carácter) del texto. No sólo no concuerdan, sino que se constata que la P no aparece en *ninguna parte* del patrón, por lo que se puede deslizarlo más allá de P. La siguiente comparación es la S



Figura 19.7 Búsqueda de cadenas de Boyer-Moore utilizando la heurística de la no concordancia.

del patrón con el quinto carácter después de P (la E de DE). Esta vez se puede deslizar el patrón hacia la derecha hasta que la D concuerde con la D del texto. Después se compara la S del patrón con la U de BÚSQUEDA, y como ésta no aparece en el patrón, se le puede deslizar cinco lugares más a la derecha. El proceso continúa hasta llegar a la D de CADENAS, punto en el que se alinea el patrón de modo que su D concuerda con la del texto y se obtiene la concordancia total. Este método conduce directamente a la posición de concordancia examinando ¡sólo siete caracteres (y cinco más para comprobar la concordancia)!

Este algoritmo del «carácter no concordante» es bastante fácil de implementar. Es una versión mejorada del método de fuerza bruta y de derecha a izquierda utilizada para inicializar un array saltar que indica, para cada carácter del alfabeto, cuántas posiciones se debe saltar en el texto si este carácter provoca una no concordancia durante la búsqueda de la cadena. Debe haber una entrada en saltar para cada carácter que pueda aparecer en el texto: para simplificar, se supone que se tiene una función índice que toma un char como argumento y devuelve 0 para los espacios en blanco e i para la i -ésima letra del alfabeto; se supone también que se dispone de una subrutina inicsaltar() que inicializa el array saltar para M caracteres que no están en el patrón y luego, para j de 0 a $M-1$, asigna a $\text{saltar}[\text{índice}(p[j])]$ el valor $M-j-1$. La implementación es directa:

```
int buscar_caracter_de_noconcordancia(char *p, char *a)
{
    int i, j, t, M = strlen(p), N = strlen(a);
    inicsaltar(p);
    for ( i = M-1, j = M-1; j > 0; i--, j--)
        while (a[i] != p[j])
        {
            t = saltar[índice(a[i])];
            i += (M-j > t) ? M-j : t;
            if (i >= N) return N;
```

```

        j = M-1;
    }
    return i;
}

```

Si la tabla saltar fuera toda 0 (lo que nunca es), esto correspondería a una versión de derecha a izquierda del método de fuerza bruta, porque la sentencia $i += M-j$ cambia el valor de i para la nueva posición de la cadena de texto (como si el patrón se moviese de izquierda a derecha a lo largo de sí mismo); entonces $j = M-1$ reasigna al puntero del patrón para prepararlo para una concordancia de derecha a izquierda, carácter a carácter. Como se acaba de presentar, la tabla saltar permite que se pueda mover el patrón a lo largo del texto tan lejos como sea posible, la mayoría de las veces M caracteres a la vez (cuando se encuentren caracteres del texto que no estén en el patrón). Para el patrón DENAS, el valor de saltar para S sería 0, para A sería 1, para N sería 2, para E sería 3, para D sería 4, y para todas las otras letras sería 5. Así, por ejemplo, cuando se encuentra una D durante una exploración de derecha a izquierda, el puntero i se incrementa en 4 de forma que el final del patrón se alinea cuatro posiciones a la derecha de D (y en consecuencia la D del patrón se alinea con la D del texto). Si hubiese más de una D en el patrón, se desearía utilizar para este cálculo la que está más a la derecha: de aquí que el array saltar se construya explorando de izquierda a derecha.

Boyer y Moore sugirieron combinar los dos métodos que se han esbozado para la exploración de derecha a izquierda, escogiendo el más grande de los dos valores de salto.

Propiedad 19.3 *La búsqueda de cadenas de Boyer-Moore nunca utiliza más de $M+N$ comparaciones de caracteres, y necesita alrededor de N/M pasos si el alfabeto no es pequeño y el patrón no es largo.*

El algoritmo es lineal en el peor caso de la misma manera que el método de Knuth-Morris-Pratt (la implementación anterior, que sólo utiliza una de las dos heurísticas de Boyer-Moore, no es lineal). El «caso medio» N/M se puede probar para varios modelos de cadenas aleatorias, pero éstos tienden a ser irreales, por lo que no se entrará en los detalles. En muchas situaciones prácticas es cierto que solamente unos pocos caracteres del alfabeto aparecen en el patrón, así que cada comparación conduce a un desplazamiento de M caracteres, lo que da el resultado señalado.■

Claro está, el algoritmo del «carácter no concordante» no ayuda mucho en el caso de cadenas binarias, porque sólo hay dos clases de caracteres que puedan causar la no concordancia (y posiblemente estén ambas en el patrón). Sin embargo, los bits se pueden agrupar para formar «caracteres» que se pueden utilizar exactamente como se vio antes. Si se toman b bits a la vez, entonces se necesita una tabla saltar con 2^b entradas. El valor de b se debe escoger lo

suficientemente pequeño para que esta tabla no sea demasiado grande, pero también lo suficientemente grande como para que la mayoría de las series de b bits del texto no estén en el patrón. Específicamente, hay $M - b + 1$ secciones diferentes de b bits en el patrón (comenzando cada una en una posición de bit desde 1 hasta $M - b + 1$), y se desea que $M - b + 1$ sea significativamente menor que 2^b . Por ejemplo, si se toma b aproximadamente igual a $\lg(4M)$, la tabla saltar estará llena en más de sus tres cuartas partes con M entradas. También b debe ser menor que $M/2$, puesto que de lo contrario se podría ocultar por completo el patrón si se dividiera en dos series de texto de b bits.

Algoritmo de Rabin-Karp

Una aproximación de fuerza-bruta al algoritmo de búsqueda de cadenas que no se ha considerado anteriormente sería explotar una gran memoria tratando cada posible serie de M caracteres del texto como si fuera una clave de una tabla de dispersión estándar. Pero no es necesario mantener una tabla de dispersión completa, puesto que el problema se plantea de forma que sólo se busque una clave; todo lo que se necesita hacer es evaluar la función dispersión para cada una de las posibles series de M caracteres del texto y verificar si es igual a la función de dispersión del patrón. El problema con este método es que parece tan difícil calcular la función de dispersión para M caracteres del texto como verificar si éstos son iguales al patrón. Rabin y Karp encontraron una forma fácil de evitar esta dificultad para la función de dispersión que se utilizó en el Capítulo 16: $h(k) = k \bmod q$, donde q (el tamaño de la tabla) es un gran entero primo. En este caso no se almacena nada en la tabla, por lo que se puede tomar un q muy grande.

El método se basa en calcular la función de dispersión para la posición i del texto conociendo su valor para la posición $i - 1$, de donde se desprende directamente una formulación matemática. Se supone que se transforman los M caracteres en números agrupándolos en una palabra en lenguaje de máquina, que podría tratarse como un entero. Esto equivale a escribir los caracteres como números en un sistema de base d , donde d es el número de caracteres posibles. El número que corresponde a $a[i]...a[i + M - 1]$ es entonces

$$x = a[i]d^{M-1} + a[i + 1]d^{M-2} + \dots + a[i + M - 1]$$

y se puede suponer que se conoce el valor de $h(x) = x \bmod q$. Pero un desplazamiento de una posición a la derecha en el texto corresponde a reemplazar x por

$$(x - a[i]d^{M-1})d + a[i + M].$$

Una propiedad fundamental de la operación mod es que si se toma el resto al

dividir por q después de cada operación aritmética (para mantener pequeños los números con los que se está tratando), se obtiene la misma respuesta que si se hubieran realizado todas las operaciones aritméticas; luego se toma el resto al dividir por q .

Esto conduce a un algoritmo muy simple de reconocimiento de patrones cuya implementación se presenta a continuación. Este programa supone la misma función `indice` anterior, pero se utiliza $d=32$ por razones de eficacia (las multiplicaciones se pueden implementar como desplazamientos).

```
const int q = 33554393;
const int d = 32;
int busquedaRK(char *p, char *a)
{
    int i, dM = 1, h1 = 0, h2 = 0;
    int M = strlen(p), N = strlen(a);
    for (i = 1; i < M; i++) dM = (d*dM) % q;
    for (i = 0; i < M; i++)
    {
        h1 = (h1*d+indice(p[i])) % q;
        h2 = (h2*d+indice(a[i])) % q;
    }
    for (i = 0; h1 != h2; i++)
    {
        h2 = (h2+d*q-indice(a[i])*dM) % q;
        h2 = (h2*d+indice(a[i+M])) % q;
        if (i > N-M) return N;
    }
    return i;
}
```

El programa calcula primero el valor de dispersión $h1$ para el patrón y el valor $h2$ para los primeros M caracteres del texto. (También calcula el valor de $d^M-1 \bmod q$ en la variable dM .) A continuación recorre la cadena de texto, utilizando la técnica anterior de calcular para cada i la función de dispersión para los M caracteres que comienzan en la posición i y comparar cada nuevo valor de dispersión con $h1$. El número primo q se escoge tan grande como se pueda, pero lo suficientemente pequeño como para que $(d+1)*q$ no provoque un desbordamiento: esto requiere menos operaciones `%` que si se utiliza el mayor número primo representable. (Se añade un $d*q$ extra durante el cálculo de $h2$ para asegurarse de que todo queda positivo y por tanto el operador `%` funciona como es debido.)

Propiedad 19.4 *Es muy probable que el método de Rabin-Karp sea lineal.*

Evidentemente este algoritmo emplea un tiempo proporcional a $N + M$, pero es preciso señalar que en realidad sólo encuentra una posición del texto que tenga el mismo valor de dispersión que el patrón. Para asegurarse de que se ha encontrado una concordancia real, se debe hacer una comparación directa de ese texto con el patrón. Sin embargo, la utilización de un valor de q muy grande, que es posible por los cálculos de % y por el hecho de que no se necesita conservar la tabla de dispersión, hace muy poco probable que se produzca una colisión. En teoría, este algoritmo podría necesitar $O(NM)$ pasos en el (casi imposible) peor caso, pero en la práctica se puede confiar en que tomará alrededor de $N + M$ pasos. ■

Búsquedas múltiples

Los algoritmos que se han presentado están todos orientados hacia un problema específico de búsqueda de cadenas: encontrar una ocurrencia de un patrón dado en una cadena de texto dada. Si la misma cadena de texto va a ser objeto de muchas búsquedas de patrones, entonces merecería la pena hacer algún procedimiento sobre la cadena para hacer más eficaces las búsquedas posteriores.

Si hay un gran número de búsquedas, el problema de la búsqueda de cadenas puede considerarse como un caso particular del problema general de búsqueda que se estudió en la sección anterior. Se trata simplemente la cadena de texto como N «claves» superpuestas, la i -ésima clave definida como $a[i], \dots, a[N]$, es decir la cadena de texto completa que comienza en la posición i . Por supuesto, no se manipularán las propias claves, sino los punteros sobre ellas: cuando se necesite comparar las claves i y j se hacen comparaciones carácter a carácter comenzando por las posiciones i y j de la cadena de texto. (Si se utiliza un carácter «centinela» final, mayor que todos los otros caracteres, una de las claves siempre es mayor que la otra.) Entonces la dispersión, el árbol binario y los otros algoritmos de la sección anterior se pueden utilizar directamente. Primero, se construye una estructura completa a partir de la cadena de texto, y luego se pueden llevar a cabo búsquedas eficaces para patrones particulares.

Es preciso realizar muchos detalles cuando se aplican de esta forma los algoritmos de búsqueda a la búsqueda de cadenas: la intención es señalar que se trata de una opción viable para algunas aplicaciones de búsqueda de cadenas. Cada situación se acompañará de métodos diferentes más o menos apropiados para diferentes situaciones. Por ejemplo, si las búsquedas son siempre de patrones de la misma longitud, una tabla de dispersión construida con una sencilla exploración, como en el método de Rabin-Karp, dará como media tiempos de búsqueda constantes. Por el contrario, si los patrones son de longitud variable, entonces alguno de los métodos basados en árboles podría ser más apropiado. (Patricia se adapta especialmente a este tipo de aplicación.)

Otras variantes del problema pueden hacerlo bastante más difícil y conducir

a métodos drásticamente diferentes, como podrá verse en los dos próximos capítulos.

Ejercicios

1. Implementar un algoritmo de reconocimiento de patrones de fuerza bruta que explore el patrón de derecha a izquierda.
2. Obtener la tabla prox para el algoritmo de Knuth-Mooris-Pratt para el patrón AAAAAAAA.
3. Obtener la tabla prox para el algoritmo de Knuth-Mooris-Pratt para el patrón ABRACADABRA.
4. Dibujar una máquina de estados finitos capaz de encontrar el patrón ABRACADABRA.
5. ¿Cómo se efectuaría una búsqueda en un archivo de texto de una cadena de 50 espacios en blanco consecutivos?
6. Obtener la tabla saltar de derecha a izquierda para la exploración de derecha a izquierda del patrón ABRACADABRA.
7. Construir un ejemplo para el que la exploración del patrón de derecha a izquierda (aplicando solamente la heurística de la no concordancia) tenga un mal rendimiento.
8. ¿Cómo se modificaría el algoritmo de Rabin-Karp para buscar un determinado patrón con la condición adicional de que el carácter central sea un «comodín» (es decir, que pueda concordar con cualquier carácter)?
9. Implementar una versión del algoritmo de Rabin-Karp para buscar patrones en un texto de dos dimensiones. Se supondrá que tanto el patrón como el texto son rectángulos de caracteres.
10. Escribir programas para generar una cadena de texto aleatoria de 1.000 bits y después encontrar todas las ocurrencias de los últimos k bits en cualquier lugar de la cadena, para $k = 5, 10, 15$. (Para diferentes valores de k pueden ser apropiados métodos diferentes.)

Reconocimiento de patrones

A menudo es deseable efectuar búsquedas de cadenas sobre la base de una pequeña información sobre el patrón a buscar. Por ejemplo, los usuarios de un editor de texto quizás quisieran especificar sólo una parte del patrón, o especificar un patrón que permita una concordancia con varias palabras diferentes, o especificar que se debe ignorar cualquier número de ocurrencias de determinados caracteres. En este capítulo se estudiará cómo se puede hacer eficazmente el *reconocimiento de patrones* de este tipo.

Los algoritmos del capítulo anterior tienen tal vez una dependencia fundamental de la especificación completa del patrón, por lo que hay que considerar otros métodos. Los mecanismos básicos que se verán permiten disponer de herramientas muy poderosas de búsqueda de cadenas, capaces de reconocer complicados patrones de M caracteres en cadenas de texto de N caracteres en un tiempo proporcional a MN^2 en el peor caso, y mucho más rápidamente en aplicaciones típicas.

En primer lugar hay que desarrollar una forma de describir los patrones: un «lenguaje» que pueda utilizarse para especificar, de forma rigurosa, los tipos de problemas de búsqueda parcial de cadenas que se han sugerido anteriormente. Este lenguaje implicará operaciones primitivas más poderosas que la simple operación de «verificar si el i -ésimo carácter de la cadena de texto concuerda con el j -ésimo carácter del patrón» utilizada en el capítulo anterior. En este capítulo se considerarán tres operaciones básicas en términos de un tipo imaginario de máquina capaz de buscar patrones en una cadena de texto. El algoritmo de reconocimiento de patrones será una forma de simular el funcionamiento de este tipo de máquina. En el próximo capítulo se verá cómo pasar de una especificación de patrón, que el usuario emplea para describir su tarea de búsqueda de cadenas, a la especificación de máquina que realmente emplea el algoritmo para llevar a cabo la búsqueda.

Como se verá, la solución que se desarrolla para resolver el problema del reconocimiento de patrones está íntimamente relacionada con ciertos procesos fundamentales de la informática. Por ejemplo, el método que se utilizará en el

programa para llevar a cabo la tarea de búsqueda de una cadena subtendida por la descripción de un determinado patrón es análogo al método utilizado por el sistema C++ para efectuar una operación de cálculo de un determinado programa en C++.

Descripción de patrones

En este capítulo se considerarán las descripciones de patrones constituidos por símbolos relacionados por las tres operaciones fundamentales siguientes:

- (i) *Concatenación*. Ésta es la operación utilizada en el capítulo anterior. Si dos caracteres son adyacentes en el patrón, entonces hay concordancia si y sólo si los dos mismos caracteres son adyacentes en el texto. Por ejemplo, AB significa A seguido de B.
- (ii) *Unión (Or)*. Ésta es la operación que permite especificar alternativas en el patrón. Si se tiene un *or* entre dos caracteres, hay concordancia si y sólo si uno de los dos caracteres figura en el texto. Se representa esta operación con el símbolo + y utilizando los paréntesis para combinarlo con la concatenación en situaciones arbitrariamente complejas. Por ejemplo A+B significa «A o B»; C(AC+B)D significa «CACD o CBD»; y (A+C)(B+C)D significa «ABD o CBD o ACD o CCD».
- (iii) *Clausura*. Esta operación permite que algunas partes del patrón se puedan repetir arbitrariamente. Si se aplica la clausura a un símbolo, entonces hay concordancia si y sólo si el símbolo aparece cualquier número de veces (incluyendo 0). La clausura se representará poniendo un * después del carácter o grupo entre paréntesis que se quiere repetir. Por ejemplo, AB* concuerda con las cadenas que consisten en una A seguida de cualquier número de B, mientras que (AB)* concuerda con las cadenas que consisten en repeticiones de la serie AB.

Una cadena de símbolos construida por medio de estas tres operaciones se denomina una *expresión regular*. Cada expresión regular describe muchos patrones de texto. El objetivo es desarrollar un algoritmo que determine si alguno de los patrones descrito por una expresión regular aparece dentro de una determinada cadena de texto.

Se concentrará la atención en la concatenación, la unión y la clausura con vistas a mostrar los principios básicos del desarrollo de algoritmos para el reconocimiento de patrones descritos por expresiones regulares. Por conveniencia, en los sistemas reales normalmente se hacen varias adiciones. Por ejemplo, -A puede significar «concuerda con cualquier carácter *excepto* con A». Esta operación *not* es la misma que un *or* de todos los caracteres diferentes de A, pero es mucho más fácil de utilizar. De modo similar, "?" significa «concuerda con cualquier letra». De nuevo, esto es evidentemente mucho más compacto que un gran *or*. Entre los otros ejemplos de símbolos adicionales que facilitan la es-

pecificación de grandes patrones figuran los símbolos de concordancia con el comienzo o el final de una línea, con una letra o un número cualquiera, etcétera.

Estas operaciones pueden ser marcadamente descriptivas. Por ejemplo, el patrón descrito por $?*(ie + ei)?*$ concuerda con todas las cadenas que tienen un *ie* o un *ei* en ellas (¡posiblemente a causa de una falta de ortografía!); el patrón $(1+01)*(0+1)$ describe todas las cadenas de 0 y 1 que no tienen dos 0 consecutivos. Evidentemente hay muchas descripciones diferentes de patrones para describir las mismas cadenas: se debe intentar especificar descripciones de patrones sucintas, al igual que se intenta escribir algoritmos eficaces.

El algoritmo de reconocimiento de patrones que se va a examinar puede verse como una generalización del método de búsqueda de cadenas por fuerza bruta de izquierda a derecha (el primer método que se vio en el Capítulo 19). El algoritmo busca la primera subcadena, empezando por la izquierda de la cadena de texto, que concuerde con la descripción del patrón. Esto lo hace explorando la cadena de texto de izquierda a derecha, comprobando la existencia, en cada posición, de una subcadena que comienza en esa posición y que concuerda con la descripción del patrón.

Máquinas de reconocimiento de patrones

Recuérdese que se puede considerar al algoritmo de Knuth-Morris-Pratt como una máquina de estados finitos, construida a partir del patrón de búsqueda que explora el texto. El método que se va a utilizar para el reconocimiento de patrones con expresiones regulares es una generalización de este proceso.

La máquina de estados finitos del algoritmo de Knuth-Morris-Pratt pasa de un estado a otro, examinando un carácter de la cadena de texto, y cambiando a un estado si hay concordancia y a otro si no la hay. Una discordancia en algún punto significa que el patrón no puede figurar en el texto precisamente en ese punto. El propio algoritmo puede representarse como una simulación de la máquina. La característica de la máquina que hace fácil su simulación es que es *determinista*: cada transición de estado se determina totalmente por el próximo carácter de entrada.

Para tratar expresiones regulares, será necesario considerar una máquina abstracta más poderosa. Debido a la operación *or*, la máquina no puede determinar cuándo aparece (o no) el patrón en un punto determinado examinando solamente un carácter. De hecho, debido a la clausura, no puede ni siquiera determinar cuántos caracteres será preciso examinar antes que se descubra una discordancia. La forma más natural de evitar estos problemas es dotar a la máquina del poder del *no determinismo*: cuando se enfrente con más de una forma de tratar de concordar con el patrón, la máquina debe «adivinar» la correcta! Esta operación parece imposible de admitir, pero se verá que es fácil de escribir un programa que simule las acciones de una tal máquina.

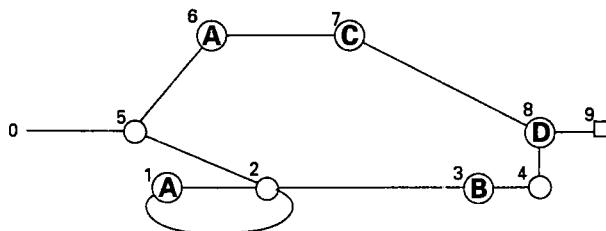


Figura 20.1 Una máquina no determinista de reconocimiento del patrón $(A^*B+AC)D$.

La Figura 20.1 muestra una máquina de estados finitos no determinista que podría utilizarse para buscar en una cadena de texto el patrón descrito por $(A^*B+AC)D$. (Los estados están enumerados según una regla que se explicará posteriormente.) Al igual que la máquina determinista del capítulo anterior, la máquina puede pasar de un estado actual etiquetado por un carácter, al estado «apuntado» por el estado actual, si puede concordar (y superar) ese carácter en la cadena de texto. Lo que hace a la máquina no determinista es que hay ciertos estados (denominados estados *nulos*) que no sólo no están etiquetados, sino que pueden «apuntar a» dos estados sucesores diferentes. (Algunos estados nulos, tal como el estado 4 del diagrama, son estados que «no operan» con una sola salida, que no afectan a la operación de la máquina, pero facilitan la implementación del programa que construye la máquina, como se verá. El estado 9 es un estado nulo sin salidas, que permite la parada de la máquina.) Cuando se encuentra en un estado nulo, la máquina puede dirigirse hacia *uno cualquiera* de los estados sucesores, independientemente de la entrada (sin superar al próximo carácter). La máquina tiene el poder de adivinar qué transición conducirá a una concordancia en la cadena de texto dada (si es que existe alguna). Se observa que no hay transiciones de «no concordancia» como en el capítulo anterior: la máquina falla en su intento de encontrar una concordancia sólo si no hay forma de adivinar una serie de transiciones que conduzcan a una concordancia.

La máquina tiene un único estado *initial* (indicado por la línea de la izquierda que no sale de ningún círculo) y un único estado *final* (el pequeño cuadrado de la derecha). Cuando se parte de un estado inicial, la máquina debe ser capaz de «reconocer» cualquier cadena descrita por el patrón leyendo caracteres y cambiando de estado de acuerdo con las reglas, hasta llegar al «estado final». Como la máquina tiene el poder del no determinismo, puede adivinar la serie de cambios de estados que pueden conducir a la solución. (Pero cuando se trata de simular esta máquina en una computadora estándar, se debe probar con todas las posibilidades.) Por ejemplo, para determinar si la descripción de patrón $(A^*B+AC)D$ puede figurar en la cadena de texto

CDAABCAAABDDACDAAC

la máquina indicaría inmediatamente un fallo si se comenzara por el primer o

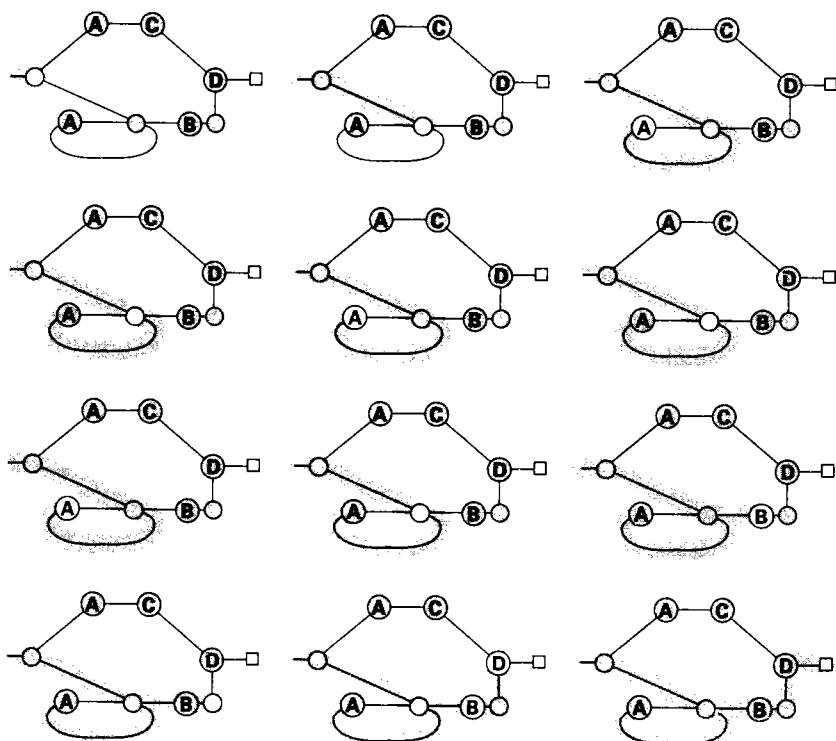


Figura 20.2 Reconocimiento de AAABD.

segundo carácter; trabajaría algo más para informar de un fallo si comenzara por los dos caracteres siguientes; indicaría inmediatamente un fallo al comenzar por el quinto o sexto carácter; y acertaría la serie de transiciones de estados que se muestra en la Figura 20.2 para reconocer AAABD, si se comienza por el séptimo carácter.

Se puede construir la maquina para una expresión regular construyendo máquinas parciales para partes de la expresión y definiendo las reglas de unión de dos máquinas parciales para formar una más grande para cada una de las tres operaciones: concatenación, *or* y clausura.

Se comienza por construir la máquina trivial para reconocer un carácter específico. Es práctico escribir esto como una máquina de dos estados, con un estado inicial (que deberá reconocer el carácter) y un estado final, como se muestra en la Figura 20.3.

Para construir la máquina para la concatenación de dos expresiones a partir de las máquinas de sus expresiones individuales, es suficiente con mezclar el estado final de la primera con el estado inicial de la segunda, como se muestra en la Figura 20.4.

De forma similar, la máquina para la operación *or* se construye añadiendo



Figura 20.3 Máquina de dos estados para reconocer un carácter.

un nuevo estado nulo que apunte a los dos estados iniciales y haciendo que uno de los estados finales apunte al otro, el cual pasa a ser el estado final de la máquina unión, como se muestra en la Figura 20.5.

Finalmente, la máquina para la operación de clausura se construye convirtiendo el estado final en estado inicial y haciéndolo apuntar hacia el antiguo estado inicial, así como sobre el nuevo estado final. Esto se muestra en la Figura 20.6.

Aplicando sucesivamente estas reglas se puede construir la máquina correspondiente a cualquier expresión regular. Los estados de la máquina de los ejemplos anteriores se han numerado en el orden de construcción, explorando el patrón de izquierda a derecha, por lo que se puede seguir fácilmente el proceso de construcción de la máquina. Nótese que se tiene una máquina elemental de dos estados por cada letra de la expresión regular y que cada + y * entrañan la crea-

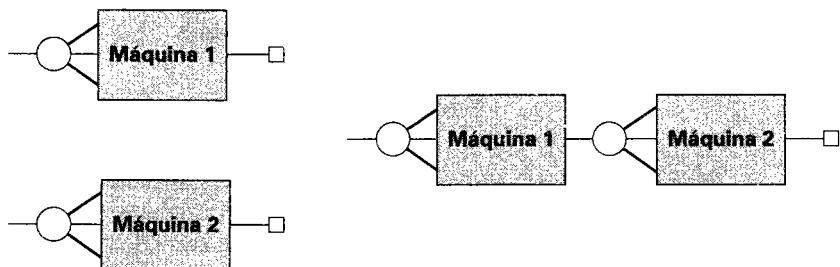


Figura 20.4 Construcción de una máquina de estados: concatenación.

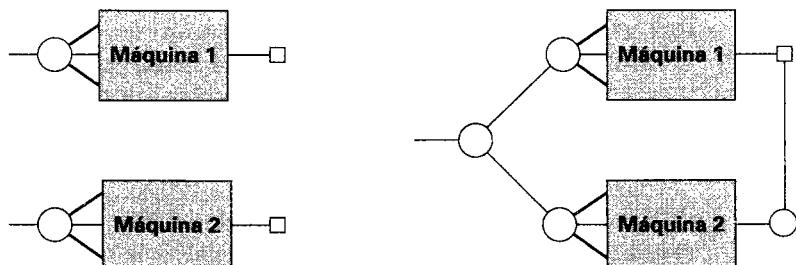


Figura 20.5 Construcción de una máquina de estados: or.

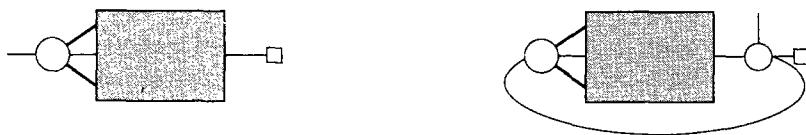


Figura 20.6 Construcción de una máquina de estados: clausura.

ción de un estado (la concatenación provoca la desaparición de uno), por lo que el número de estados es inferior a dos veces el número de caracteres de la expresión regular.

Representación de la máquina

Las mencionadas máquinas no deterministas se construirán utilizando sólo las reglas de composición esbozadas anteriormente, y se podrá aprovechar su estructura tan simple para manipularlas de forma directa. Por ejemplo, de un estado cualquiera sale un máximo de dos líneas. De hecho, hay solamente dos tipos de estados: los etiquetados por un carácter del alfabeto de entrada (de los que sale una sola línea) y los no etiquetados (nulos) (de los que salen dos o menos líneas). Esto significa que estas máquinas se pueden representar con muy poca información por nodo. Puesto que a menudo se desea acceder a los estados por su número, la forma más conveniente de organización para la máquina es una representación por array. Se utilizarán tres arrays paralelos carac, prox1, y prox2, indexados por estado, para representar y tener acceso a la máquina. Sería posible lograrlo con dos tercios de este espacio de memoria, puesto que cada estado realmente sólo utiliza dos partes significativas de información, pero no se hará uso de estas mejoras para ganar en claridad y también porque en definitiva es probable que las descripciones de los patrones no sean muy largas.

La máquina anterior se puede representar como indica la Figura 20.7. Las entradas indexadas por estado pueden interpretarse como instrucciones para la máquina no determinista de la forma «Si está usted en estado y ve ca-

	0	1	2	3	4	5	6	7	8	9
carac		A		B			A	C	D	
prox1	5	2	3	4	8	6	7	8	9	0
prox2	5	2	1	4	8	2	7	8	9	0

Figura 20.7 Representación por array de la máquina de la Figura 20.1.

rac[estado], entonces lea el carácter y vaya al estado prox1[estado] (o al prox2[estado])». En este ejemplo, el Estado 9 es el estado final y el estado 0 es un estado pseudo-inicial cuyos valores en los arrays prox son los números de los estados iniciales reales. Obsérvese la representación especial utilizada para los estados nulos con un sucesor (las entradas en los dos arrays prox son iguales) y para el estado final (con cero en ambas entradas de los arrays prox).

Se ha visto la forma de construir máquinas para patrones descritos por expresiones regulares y cómo representar tales máquinas con arrays. Sin embargo, escribir un programa que haga pasar de una expresión regular a la representación correspondiente por una máquina no determinista es otra cosa. En efecto, incluso escribir un programa para determinar si una expresión regular es legal es un reto para los principiantes. En el próximo capítulo se estudiará detalladamente esta operación, denominada *análisis sintáctico*. Por el momento, se supondrá que se ha hecho dicho paso, por lo que se dispone de los arrays carac, prox1, y prox2 que representan a una máquina no determinista determinada, que corresponde a la expresión regular que describe al patrón objeto de interés.

Simulación de la máquina

El último paso en el desarrollo de un algoritmo general para el reconocimiento de patrones descritos por expresiones regulares consiste en escribir un programa que de alguna forma simule la operación de una máquina no determinista de reconocimiento de patrones. La idea de escribir un programa que pueda «adivinar» la respuesta correcta parece ridícula. Sin embargo, en este caso se puede ir «memorizando» sistemáticamente *todas las posibles* concordancias, de modo que siempre se acabe por encontrar la correcta.

Una posibilidad sería desarrollar un programa recursivo que imite a la máquina no determinista (pero que trate todas las posibilidades en lugar de estar adivinando la correcta). En lugar de utilizar esta variante, se va a examinar una implementación no recursiva que expondrá los principios básicos de operación del método, guardando los estados que se están considerando en una estructura de datos algo peculiar denominada *deque* (*cola de doble extremo*).

La idea es conservar todos los estados que se podrían encontrar mientras la máquina está «mirando» el carácter de entrada. Cada uno de estos estados se procesa en su momento: los estados nulos conducen a dos estados (o menos), los estados para caracteres que no concuerdan con el carácter de entrada se eliminan y los estados para caracteres que concuerdan con el carácter de entrada conducen a nuevos estados a utilizar cuando la máquina esté examinando el *próximo* carácter de entrada. Así pues, se desea mantener una lista de todos los estados en los que la máquina no determinista podría encontrarse en un punto particular del texto. El problema es diseñar una estructura de datos apropiada para esta lista.

El procesamiento de los estados nulos parece necesitar una *pila*, puesto que

esencialmente se está posponiendo una de las dos acciones a tomar, al igual que al eliminar una recursión (por tanto, el nuevo estado se debe poner al *comienzo* de la lista para que no quede pospuesto indefinidamente). El procesamiento de los otros estados parece necesitar una *cola*, dado que no se desea examinar los estados correspondientes al próximo carácter de entrada hasta que no se termine con el carácter en curso (por tanto el nuevo estado se debería poner al *final* de la lista). En lugar de escoger entre estas dos estructuras, ¡se utilizarán las dos! Las *deques* combinan las características de las pilas y de las colas: una *deque* es una lista en la que los elementos se pueden añadir por los dos extremos. (En realidad, se utiliza una «*deque* de salida restringida», puesto que se quitan los elementos del comienzo, no del final.)

Una propiedad primordial de la máquina es que no tiene «bucle» que estén constituidos únicamente por estados nulos, puesto que de otra manera se podría caer, en una forma no determinista, en un bucle infinito. Esto implica que el número de estados de la deque, en cualquier momento, es menor que el número de caracteres de la descripción del patrón.

El programa que se presenta posteriormente utiliza una deque para simular las acciones de una máquina de reconocimiento de patrones no determinista como la que se describió con anterioridad. Mientras está examinando un carácter particular en la entrada, la máquina no determinista puede encontrarse en un número limitado de estados posibles: el programa conserva la pista de estos estados en una deque, utilizando los procedimientos *meter*, *poner* y *sacar*, parecidos a los del Capítulo 3. Se podría utilizar también una representación por array (como en la implementación de cola del Capítulo 3) o una representación por lista enlazada (como en la implementación de pila del Capítulo 3). Se omiten los detalles de la implementación.

El bucle principal del programa retira un estado de la deque y lleva a cabo la acción requerida. Si se va a hacer la concordancia con un carácter, se comprueba si éste existe en la entrada: si hay concordancia se efectúa el cambio de estado poniendo el nuevo estado al *final* de la deque (por tanto, todos los estados que implican el carácter en curso se procesan antes que los que implican el carácter siguiente). Si el estado es nulo, los dos estados posibles que se deben simular se ponen al *comienzo* de la deque. Los estados que implican el carácter en curso se guardan separadamente de aquellos que implican el próximo carácter mediante una marca *avanza*=-1 en la deque: cuando se encuentra esta marca «*avanza*», se avanza el puntero en la cadena de entrada. El bucle termina cuando se llega al final de la entrada (no se encontró una concordancia), cuando se llega al estado 0 (se encontró una concordancia legal), o cuando sólo está la marca *avanza* en la deque (no se encontró ninguna concordancia). Esto conduce directamente a la siguiente implementación:

```
const int avanza = -1;
int concordar(char *a)
{
    int n1, n2; Deque dq(100);
```

```

int j = 0, N = strlen(a), estado = prox1[0];
dq.poner(avanza);
while (estado)
{
    if (estado == avanza) { j++; dq.poner(avanza); }
    else if (carac[estado] == a[j])
        dq.poner(prox1[estado]);
    else if (carac[estado] == ' ')
    {
        pl = prox1[estado]; n2 = prox2[estado];
        dq.meter(pl);
        if (pl != p2) dq.meter(p2);
    }
    if (dq.vacio() || j==N) return 0;
    estado = dq.sacar();
}
return j;
}

```

Esta función toma como argumento el puntero a la cadena de texto *a* en la que se intenta encontrar una concordancia, utilizando la máquina no determinista que representa al patrón a través de los arrays *carac*, *prox1* y *prox2* descritos anteriormente. Esta función devuelve la longitud de la subcadena inicial más corta que concuerda con el patrón (y 0 si no hay concordancia). Por conveniencia, se supone que el último carácter de la cadena de texto *a* es un carácter centinela único que no se repite en ningún otro lugar del array *carac* que representa al patrón.

La Figura 20.8 muestra el contenido de la deque cada vez que se elimina un

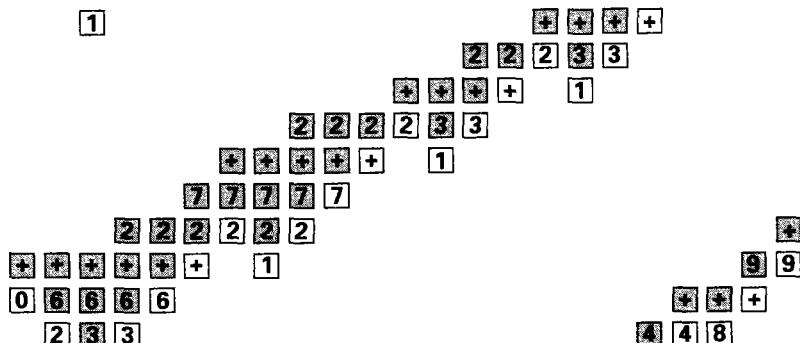


Figura 20.8 Contenido de la deque durante el reconocimiento de AAABD.

estado cuando la máquina ejemplo se pone a trabajar con la cadena de texto AAABD. Este diagrama supone una representación por array, como la utilizada para las colas del Capítulo 3: se utiliza un signo de suma para representar avanza. Cada vez que la marca avanza alcanza el frente de la deque (parte inferior del diagrama), el puntero j avanza hacia el siguiente carácter del texto. Así pues, se comienza con el estado 5 mientras se explora el primer carácter del texto (la primera A). El estado 5 conduce a los estados 2 y 6; después el estado 2 conduce a los estados 1 y 3, los cuales necesitan leer el mismo carácter y se encuentran al comienzo de la deque. Después el estado 1 conduce al estado 2, pero al final de la deque (para el próximo carácter de entrada). El estado 3 conduce a otro estado sólo mientras se está explorando una B; por lo tanto se ignora mientras se está explorando una A. Cuando finalmente el centinela «avanza» alcanza el frente de la deque, se ve que la máquina puede estar en el estado 2 o en el estado 7 después de leer una A. El programa trata entonces los estados 2, 1, 3 y 7 mientras «está mirando» a la segunda A, para descubrir, la segunda vez que avanza llega al comienzo de la deque, que el estado 2 es la única posibilidad después de la exploración de AA. Ahora, mientras se está examinando la tercera A, las únicas posibilidades son los estados 2, 1 y 3 (la posibilidad AC ahora está excluida). Estos tres estados se tratan nuevamente, para conducir por último al estado 4 después de la exploración de AAAB. Continuando, el programa va al estado 8, pasa la D y termina en el estado final. Se ha encontrado una concordancia, pero, lo que es más importante, se han considerado todas las transiciones coherentes con la cadena de texto.

Propiedad 20.1 *La simulación del funcionamiento de una máquina de M estados para buscar patrones en un texto de N caracteres se puede hacer con menos de NM transiciones de estados en el peor caso.*

Por supuesto, el tiempo de ejecución de concordar depende muy fuertemente del patrón que se está reconociendo. Sin embargo, para cada uno de los N caracteres de entrada, parece que se procesan a lo sumo M estados de la máquina; por tanto, el tiempo de ejecución del peor caso debe ser proporcional a MN (para cada posición de comienzo en el texto). Por desgracia, esto no es cierto para concordar, tal como antes se ha implementado, porque cuando se pone un estado en la deque el programa no verifica si ya estaba allí, por lo que la deque puede contener copias duplicadas de un mismo estado. Esto puede no tener mucho efecto en aplicaciones prácticas, pero sí provocar una ejecución excesiva en algunos casos patológicos si se deja sin verificar. Por ejemplo, este problema tarde o temprano conduce a una deque con 2^{N-1} estados cuando se concuerda el patrón $(A^*A)^*B$ con una cadena de N A seguida de una B. Para evitar esto, las rutinas de la deque utilizadas por concordar deben estar implementadas de forma tal que se eviten las duplicaciones de estados en la deque (con el fin de garantizar que a lo sumo se procesarán M estados por cada carácter de entrada). Esto se puede hacer manteniendo un array indexado por estado, que indica qué estados están en la deque. Con este cambio, el número total de operaciones

necesarias para determinar si una porción de la cadena de texto está descrita por el patrón se encuentra en $O(MN^2)$.■

No todas las máquinas no deterministas se pueden simular tan eficazmente, como se verá con más detalle en el Capítulo 40, pero la utilización de una hipotética máquina simple para el reconocimiento de patrones conduce a un algoritmo bastante razonable para un problema bastante difícil. Sin embargo, para completar el algoritmo se necesita un programa que permita pasar de expresiones regulares arbitrarias a «máquinas» que se puedan interpretar con el código anterior. En el próximo capítulo se verá la implementación de un programa tal en el contexto de una presentación más general de técnicas de compilación y de análisis sintáctico.

Ejercicios

1. Obtener una expresión regular para el reconocimiento de todas las ocurrencias de una serie de cuatro (o menos) 1 consecutivos en una cadena binaria.
2. Dibujar la máquina no determinista de reconocimiento de patrones para la descripción del patrón $(A+B)^*+C$.
3. Plantear las transmisiones de estados que haría la máquina del ejercicio anterior para reconocer ABBAC.
4. Explicar cómo se modificaría la máquina no determinista para manipular la función *not*.
5. Explicar cómo se modificaría la máquina no determinista para manipular caracteres del tipo «sin importancia».
6. ¿Cuántos patrones diferentes se pueden describir por una expresión regular con M operadores *or* y ningún operador de clausura?
7. Modificar concordar para que manipule expresiones regulares con la función *not* y caracteres del tipo «sin importancia».
8. Mostrar cómo construir una descripción de un patrón de longitud M y una cadena de texto de longitud N para los que el tiempo de ejecución de concordar sea tan grande como sea posible.
9. Implementar una versión de concordar que evite el problema descrito en la demostración de la propiedad 20.1.
10. Mostrar el contenido de la *deque* cada vez que se suprime un estado al utilizar concordar para simular la máquina del ejemplo que se ha utilizado en el capítulo, con la cadena de texto ACD.

Análisis sintáctico

Se han desarrollado diversos algoritmos fundamentales para reconocer si los programas de computadora son válidos y descomponerlos de forma propicia para su posterior procesamiento. Esta operación, denominada *análisis sintáctico*, tiene aplicaciones más allá de la informática, dado que está relacionada con el estudio de la estructura del lenguaje en general. Por ejemplo, el análisis sintáctico tiene un papel fundamental en los sistemas que tratan de «entender» los lenguajes naturales (humanos) y en los de traducción de una lengua a otra. Un caso particular de interés es la transformación de un lenguaje de computadora «de alto nivel» como C++ (conveniente para el uso humano) a un lenguaje de «bajo nivel» como un ensamblador o uno de máquina (conveniente para ejecutar por computadora). Un programa que hace tales transformaciones se denomina un *compilador*. De hecho, ya se ha visto un método de análisis sintáctico, en el Capítulo 4, cuando se construyó un árbol para representar una expresión aritmética.

En el análisis sintáctico se utilizan dos metodologías generales. Los *métodos descendentes* tratan de probar si un programa es válido buscando en primer lugar las partes del programa que son válidas, y después las partes de esas partes, etc., hasta que las piezas sean lo suficientemente pequeñas como para que correspondan directamente con la cadena de entrada. Los métodos *ascendentes* van juntando piezas de la entrada de una manera estructurada formando piezas cada vez mayores hasta que se obtenga un programa válido. En general, los métodos descendentes son recursivos y los ascendentes iterativos. En general se piensa que los métodos descendentes son más fáciles de implementar y los ascendentes más eficaces. El método del Capítulo 4 era ascendente; en este capítulo se estudiará con detalle un método descendente.

El tratamiento completo de los temas referentes a los analizadores sintácticos y a la construcción de compiladores está claramente fuera del alcance de este libro. Sin embargo, mediante la construcción de un «sencillo compilador» para completar el algoritmo de reconocimiento de patrones del capítulo anterior, se estará también considerando algunos de los conceptos fundamentales

subyacentes. Primero se construirá un analizador sintáctico descendente para un lenguaje simple de descripción de expresiones regulares. Luego se modificará el analizador sintáctico para hacer un programa que traduzca expresiones regulares en máquinas de reconocimiento de patrones que puedan utilizarse por el procedimiento concordar del capítulo anterior.

La intención del capítulo es proporcionar una aproximación a los principios básicos del análisis sintáctico y la compilación, a la vez que se desarrolla un algoritmo útil de reconocimiento de patrones. Ciertamente no se podrá abordar todo lo que se trate aquí con la profundidad que se merece. El lector debe saber que pueden aparecer dificultades sutiles cuando se aplica la misma estrategia a problemas similares, y que la construcción de compiladores es un campo bastante rico con una gran variedad de métodos avanzados de aplicación en situaciones serias.

Gramáticas libres de contexto

Antes de que se pueda escribir un programa que determine si es válido un programa escrito en un cierto lenguaje, se necesita una descripción de lo que caracteriza exactamente a un programa válido. Esta descripción se denomina *gramática*: para apreciar esta terminología basta con pensar en una lengua como la del lector y reemplazar en la oración anterior «oración» por «programa» (excepto la primera vez que aparece programa!). Los lenguajes de programación se describen a menudo con un tipo particular de gramática denominada *gramática libre de contexto*. Por ejemplo, las siguientes reglas caracterizan a la gramática libre de contexto definida por el conjunto de todas las expresiones regulares válidas (como las descritas en el capítulo anterior).

```

<expresión> ::= <término> | <término> + <expresión>
<término> ::= <factor> | <factor><término>
<factor> ::= (<expresión>) | v | (<expresión>)* | v*

```

Esta gramática describe expresiones regulares como las que se utilizaron en el capítulo anterior, tales como $(1+01)^*(0+1)$ o $(A^*B+AC)D$. Cada línea de la gramática se denomina una *producción* o *regla de reemplazo*. Las producciones se componen de símbolos *terminales* (,), + y *, que son los símbolos utilizados en el lenguaje que se está describiendo (y «v», un símbolo especial, representa a cualquier letra o dígito); de símbolos *no terminales* <expresión>, <término> y <factor>, que son internos a la gramática; y de *metasímbolos* ::= y |, que sirven para describir el significado de las producciones. El símbolo ::=, que puede leerse como «*es un*», define la parte izquierda de la producción (la cadena a la izquierda del ::=) en función de los términos de la parte derecha; y el símbolo |, que puede leerse como un «*o*» (*or*), que indica alternativas de selección. Las

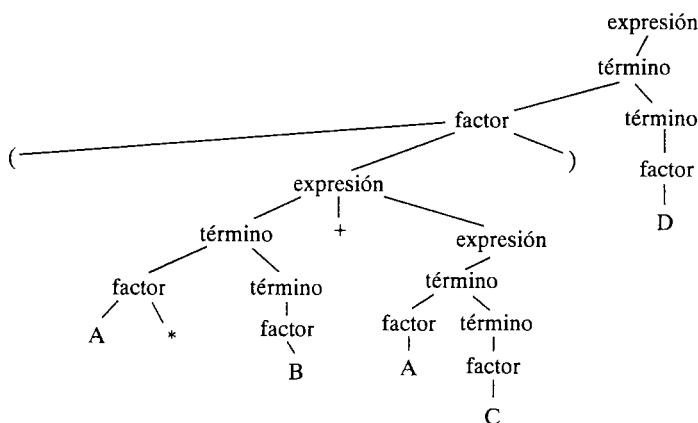


Figura 21.1 Un árbol de análisis sintáctico para $(A^*B+AC)D$.

producciones expresadas con esta concisa notación simbólica corresponden de forma simple a una descripción intuitiva de la gramática. Por ejemplo, la segunda producción de la gramática del ejemplo puede leerse como «un *<término>* es un *<factor>* o es un *<factor>* seguido de un *<término>*». Un símbolo no terminal, en este caso *<expresión>*, se *distingue* en el sentido de que una cadena de símbolos terminales pertenece al lenguaje descrito por la gramática si y sólo si hay alguna forma de utilizar las reglas de producciones para *derivar* esa cadena del no terminal *distinguido* reemplazando (en cualquier número de pasos) un símbolo no terminal por alguna de las cláusulas *or* de la parte derecha de la producción de ese símbolo no terminal.

Una forma natural de describir el resultado de este proceso de derivación es un *árbol de análisis sintáctico*: un diagrama de la estructura gramatical completa de la cadena que se está analizando. Por ejemplo, el árbol de análisis sintáctico de la Figura 21.1 muestra que la cadena $(A^*B+AC)D$ pertenece al lenguaje descrito por la gramática anterior. A veces se utilizan árboles de análisis sintáctico como éstos para descomponer una oración de un lenguaje natural, en sujeto, verbo, objeto, etcétera.

La función principal de una analizador sintáctico es aceptar las cadenas que se puedan derivar y rechazar las que no se puedan, intentando construir un árbol de análisis sintáctico para una cadena dada. Esto es, el analizador sintáctico puede *reconocer* si una cadena pertenece al lenguaje descrito por la gramática determinando si existe o no un árbol de análisis sintáctico para esa cadena. Los analizadores sintácticos descendentes hacen esto construyendo el árbol comenzando por el símbolo no terminal *distinguido* en la raíz y descendiendo hacia la cadena que se debe reconocer situada en el fondo del árbol. Los analizadores sintácticos ascendentes lo hacen comenzando por la cadena del fondo del árbol y ascendiendo hasta llegar al no terminal *distinguido* de la raíz. Como se verá,

si la semántica de las cadenas a reconocer implica un procesamiento posterior, entonces el analizador sintáctico puede convertir las cadenas en una representación interna que facilite tal procesamiento.

Otro ejemplo de una gramática libre de contexto se puede encontrar en el apéndice del libro *The C++ Programming Language* que describe los programas que son válidos en C++. Los principios considerados en esta sección para el reconocimiento y empleo de expresiones lícitas se aplican directamente a la compleja tarea de compilar y ejecutar programas en C++. Por ejemplo, la gramática siguiente describe un subconjunto muy pequeño de C++, las expresiones aritméticas que contienen sumas y multiplicaciones:

```
<expresión> ::= <término> | <término> + <expresión>
<término> ::= <factor> | <factor>* <término>
<factor> ::= (<expresión>) | v
```

Estas reglas describen de una manera formal lo que se aceptó como válido en el Capítulo 4: son reglas que especifican las expresiones aritméticas «válidas». De nuevo, *v* es un símbolo especial que representa cualquier letra, pero en esta gramática las letras pueden representar variables con valores numéricos. $A+(B*C)$ y $A*((B+C)*(D+E))+F$ son ejemplos de cadenas válidas para esta gramática. Ya se vio en el Capítulo 4 un árbol de análisis sintáctico para la segunda de estas cadenas, pero dicho árbol no correspondía a la gramática anterior; por ejemplo, no incluía explícitamente los paréntesis.

Tal como se han definido las cosas hasta aquí, algunas cadenas son perfectamente válidas como expresiones aritméticas y como expresiones regulares. Por ejemplo, $A*(B+C)$ puede significar «sumar B a C y multiplicar el resultado por A» o «tomar un número cualquiera de A seguido de una B o de una C». Este punto evidencia el hecho obvio de que verificar si una cadena es válida es una cosa, pero comprender su significado es otra diferente. Se volverá sobre este tema una vez que se haya visto cómo analizar una cadena para verificar si está descrita o no por una cierta gramática.

Cada expresión regular es por sí misma un ejemplo de gramática libre de contexto: cualquier lenguaje que se puede describir con una expresión regular también se puede describir con una gramática libre de contexto. La recíproca no es cierta: por ejemplo, el concepto de paréntesis «equilibrados» no se puede reproducir por medio de una expresión regular. Otros tipos de gramáticas pueden describir lenguajes que las gramáticas libres de contexto no pueden. Por ejemplo, las gramáticas *dependientes del contexto* o *sensibles al contexto* son idénticas a las anteriores excepto que la parte izquierda de las producciones no tiene que ser de sólo un no terminal. Las diferencias entre clases de lenguajes y la jerarquía de las gramáticas que los describen se han estudiado muy cuidadosamente constituyendo una magnífica teoría que se ubica en el corazón de la ciencia informática.

Análisis descendente

Algunos métodos de análisis sintáctico utilizan la recursión para reconocer las cadenas del lenguaje descritas exactamente como se especificaron por la gramática. De forma más simple, la gramática es una especificación tan completa del lenguaje que se puede poner directamente en forma de programa!

Cada producción corresponde a un procedimiento con el mismo nombre que el no terminal de la parte izquierda. Los no terminales de la parte derecha corresponden a llamadas (posiblemente recursivas) a procedimientos; los terminales corresponden a la exploración de la cadena de entrada. Por ejemplo, el procedimiento siguiente es parte de un analizador sintáctico descendente para la gramática de expresiones regulares:

```
expresion()
{
    termino();
    if (p[j] == '+')
        { j++; expresion(); }
}
```

Una cadena p contiene la expresión regular que se está analizando, con un índice j que apunta al carácter que se está examinando actualmente. Para analizar una expresión regular dada p, se pone j en 0 y se llama a expresion. Si resulta que j llega a ser M, entonces la expresión regular pertenece al lenguaje descrito por la gramática. Si no, se verá a continuación cómo tratar las distintas situaciones de error. La primera acción de expresion es llamar a termino, cuya implementación es algo más complicada:

```
termino()
{
    factor();
    if ((p[j] == '(') || letra(p[j])) termino();
```

Una implementación directa de la gramática haría que termino llamara primero a factor y luego a termino. Esto está evidentemente condenado al fracaso porque no hay forma de salir de termino: este programa caería en un bucle infinito de llamadas recursivas. (Tales bucles tienen efectos muy molestos en muchos sistemas.) La implementación anterior evita esto comprobando en primer lugar la cadena de entrada para decidir si se debe o no llamar a termino. Lo primero que termino hace es llamar a factor, que es el único de los procedimientos que puede detectar una no concordancia con la cadena de entrada. Por la gramática se sabe que cuando se llama a factor, el carácter actual de la

entrada debe ser o un «» o una letra (representada por v). Este proceso de verificar el próximo carácter de la entrada, sin incrementar j , para decidir qué hacer, se denomina *examen por anticipado (anticipación)*. En algunas gramáticas esto no es necesario, pero en otras puede ser el caso que se necesite más de una anticipación.

La implementación de factor se obtiene ahora directamente de la gramática. Si el carácter que se está explorando no es «» o una letra, se llama a un procedimiento error para manipular la condición de error:

```
factor()
{
    if (p[j] == '(')
    {
        j++; expresion();
        if (p[j] == ')') j++; else error();
    }
    else if (letra(p[j])) j++; else error();
    if (p[j] == '*') j++;
}
```

Otra condición de error se produce cuando falta un «».

Las funciones expresion, termino y factor son evidentemente recursivas; de hecho están tan interrelacionadas que no hay forma de escribir las de forma tal que se pueda declarar cada función antes de llamarla (esto representa una dificultad en ciertos lenguajes de programación).

El árbol de análisis sintáctico de una cadena dada proporciona la estructura de las llamadas recursivas durante el análisis sintáctico. La Figura 21.2 muestra sucesivamente las tres operaciones anteriores cuando p contiene la cadena $(A^*B+AC)D$ y se llama a expresion con $j=1$. Excepto para el signo +, toda la «exploración» se efectúa en factor. Para mayor legibilidad, los caracteres que recorre factor, excepto los paréntesis, se inscriben en la misma línea que la llamada a factor.

Se anima al lector a relacionar este proceso con la gramática y el árbol de la Figura 21.1. Este proceso corresponde a recorrer el árbol en orden previo, aunque la correspondencia no es exacta porque la estrategia de anticipación busca esencialmente cambiar la gramática. Puesto que se parte de la raíz del árbol y se trabaja hacia abajo, es evidente el origen del nombre «descendente». Tales analizadores sintácticos también se denominan *analizadores sintácticos recursivo descendentes* porque recorren el árbol recursivamente.

Esta estrategia descendente no funciona con todas las gramáticas libres de contexto posibles. Por ejemplo, si la producción $\langle \text{expresión} \rangle ::= v \mid \langle \text{expresión} \rangle + \langle \text{termino} \rangle$ se llevara mecánicamente a C++, se obtendría el siguiente resultado indeseable:

```

mala_expresion()
{
    if (letra(p[j])) j++;
    else
    {
        mala_expresion();
        if (p[j] == '+') { j++; termino(); }
        else error();
    }
}

```

Si este procedimiento se llamara cuando en $p[j]$ hubiera un valor diferente de una letra (como en el ejemplo, para $j=1$) caería en un bucle recursivo infinito. Evitar tales bucles es una de las principales dificultades en la implementación de analizadores recursivos descendentes. En *termino*, se utiliza una anticipación para evitar un bucle de esta naturaleza; en este caso una buena solución consiste en invertir la gramática y decir $\langle\text{termino}\rangle + \langle\text{expresión}\rangle$ en lugar de $\langle\text{expresión}\rangle + \langle\text{termino}\rangle$. La ocurrencia de un no terminal como primer elemento de la parte derecha de una producción que tiene en la parte izquierda el mismo no terminal se denomina *recursividad izquierda*. De hecho, el problema

```

expresion
termino
factor
(
expresion
termino
    factor A *
termino
    factor B
+
expresion
termino
    factor A
termino
    factor C
)
termino
factor D

```

Figura 21.2 Análisis sintáctico de $(A^*B+AC)D$.

es más sutil, porque la recursión izquierda puede aparecer indirectamente, por ejemplo en las producciones $\langle \text{expresión} \rangle ::= \langle \text{término} \rangle$ y $\langle \text{término} \rangle ::= v \mid \langle \text{expresión} \rangle + \langle \text{término} \rangle$. Los analizadores sintácticos recursivos descendentes no funcionan con tales gramáticas; es preciso transformarlos en gramáticas equivalentes sin la recursión izquierda, o bien utilizar algún otro método de análisis sintáctico. En general, hay una conexión muy íntima y ampliamente estudiada entre los analizadores sintácticos y las gramáticas que éstos reconocen. La elección de la técnica de análisis sintáctico depende a menudo de las características de la gramática que se va a analizar.

Análisis sintáctico ascendente

Aunque hay varias llamadas recursivas en los programas anteriores, es un ejercicio instructivo eliminar sistemáticamente la recursión. Se vio en el Capítulo 5 que cada llamada a procedimiento se puede reemplazar por meter (*push*) en una pila y cada retorno de procedimiento por sacar (*pop*) de la pila (reproducido lo que hace el sistema C++ para implementar la recursión). Hay que recordar que una razón para hacer esto es que muchas llamadas que parecen recursivas en realidad no lo son. Cuando una llamada a procedimiento es la última acción de un procedimiento, entonces se puede utilizar un simple *goto*. Esto convierte a *expresion* y a *termino* en simples bucles que se pueden fusionar y combinar con *factor* para producir un procedimiento único con una sola llamada verdaderamente recursiva (la llamada a *expresion* dentro de *factor*).

Este punto de vista conduce directamente a una forma bastante simple de comprobar si una expresión regular es válida. Una vez que se han eliminado todas las llamadas a procedimientos, se ve que cada símbolo terminal se recorre sólo cuando se encuentra. El único procesamiento real consiste en comprobar cuándo existe un paréntesis derecho que concuerde con cada paréntesis izquierdo, si cada «+» está seguido por una letra o un «(», y si cada «*» sigue a una letra o a un «)».

Esto es, comprobar si una expresión regular es válida es en esencia equivalente a la comprobación de paréntesis equilibrados. Esto se puede implementar sencillamente con un contador, inicializado a 0, que se incrementa cuando se encuentra un paréntesis izquierdo y se decrementa cuando se encuentra un paréntesis derecho. Si el contador es cero al finalizar la expresión, y los «+» y los «*» dentro de la misma cumplen con las condiciones que se acaban de mencionar, entonces la expresión es válida.

Por supuesto, el análisis sintáctico es algo más que comprobar si la cadena de entrada es válida: el objetivo principal es construir el árbol de análisis sintáctico (incluso de una forma implícita, como en el analizador sintáctico descendente) para llevar a cabo otros procesamientos. Parece posible hacer lo mismo en programas que tengan esencialmente la misma estructura que el verificador de paréntesis que se acaba de describir. Un tipo de analizador que funciona de

esta manera es el denominado *analizador sintáctico desplazamiento-reducción*. La idea es utilizar una pila que almacene los símbolos terminales y no terminales. Cada paso del análisis sintáctico es o bien un paso *desplazamiento*, en el que se pone en la pila el siguiente carácter de entrada, o un paso *reducción*, en el que se concuerdan los caracteres de la cima de la pila con la parte derecha de alguna regla de producción de la gramática y se «reducen a» (se reemplazan por) el no terminal de la parte izquierda de la misma producción. (La dificultad principal al construir un analizador sintáctico desplazamiento-reducción es decidir cuándo se desplaza y cuándo se reduce. Esta puede ser una decisión compleja, dependiendo de la gramática.) Tarde o temprano todos los caracteres de entrada tienen que haber pasado a la pila y, también al fin y al cabo, la pila se reduce a un único símbolo no terminal. Los programas de los Capítulos 3 y 4, que construyen un árbol de análisis sintáctico a partir de una expresión infija, transformándola primero en una expresión postfija, son un ejemplo de un analizador sintáctico de este tipo.

En general el análisis sintáctico ascendente se considera como el método a elegir para los lenguajes de programación. Hay una extensa literatura sobre el desarrollo de analizadores sintácticos para grandes gramáticas, del tipo que se necesita para describir un lenguaje de programación. Esta breve descripción sólo roza la superficie de los temas de este campo.

Compiladores

Un *compilador* puede considerarse como un programa que traduce de un lenguaje a otro. Por ejemplo, un compilador C++ traduce programas escritos en lenguaje C++ al lenguaje de máquina de una computadora determinada. Se ilustrará la forma de efectuar esta traducción continuando con el ejemplo de reconocimiento de patrones descritos por expresiones regulares. Sin embargo, ahora se desea traducir del lenguaje de las expresiones regulares a las máquinas de reconocimiento de patrones, es decir los arrays `carac`, `prox1` y `prox2` del programa concordar del capítulo anterior.

El proceso de traducción es esencialmente «uno a uno»: para cada carácter del patrón (con la excepción de los paréntesis) se desea generar un estado de la máquina de reconocimiento de patrones (una entrada de cada array). La clave está en conservar la información necesaria para llenar los arrays `prox1` y `prox2`. Para hacer esto, se convierte cada uno de los procedimientos del analizador sintáctico recursivo descendente en funciones generadoras de máquinas de reconocimiento de patrones. Cada función añadirá al final de los arrays `carac`, `prox1` y `prox2` tantos nuevos estados como sea necesario, y devolverá el índice del estado inicial de la máquina generada (el estado final será siempre la última entrada de los arrays). Así es que, por ejemplo, la siguiente función para la producción de `<expresion>` genera los estados `or` para la máquina de reconocimiento de patrones.

```

int expresion()
{
    int t1, t2, r;
    t1 = termino(); r = t1;
    if (p[j] == '+')
    {
        j++; estado++;
        t2 = estado; r = t2; estado++;
        actualiza_estado(t2, ' ', expresion(), t1);
        actualiza_estado(t2-1, ' ', estado, estado);
    }
    return r;
}

```

Esta función utiliza un procedimiento `actualiza_estado` que asigna a las entradas de los arrays `carac`, `prox1` y `prox2`, indexadas por el primer argumento, los valores proporcionados por el segundo, tercero y cuarto argumentos, respectivamente. El argumento `índice` conserva el estado «actual» de la máquina que se está construyendo: cada vez que se crea un nuevo estado se incrementa `índice`. Así pues, los índices de estados de la máquina que corresponden a una llamada a procedimiento particular varían entre el valor que tiene `índice` a la entrada del procedimiento y su valor a la salida. El índice del estado final es el valor de `índice` a la salida. (Realmente no se «crea» el estado final incrementando `índice` antes de la salida, puesto que esto facilita la «fusión» del estado final con posteriores estados iniciales, como se podrá comprobar.)

Con este convenio, es fácil comprobar (¡cuidado con la llamada recursiva!) que el programa anterior implementa la regla de composición de dos máquinas con la operación *or* según el diagrama del capítulo anterior. Primero se construye (recursivamente) la máquina para la primera parte de la expresión; luego se añaden dos nuevos estados nulos y se construye la segunda parte de la expresión. El primer estado nulo (de índice `t2-1`) es el estado final de la máquina de la primera parte de la expresión, el cual se pone en un estado «no operativo» para pasar al estado final de la máquina de la segunda parte de la expresión, como es de desechar. El segundo estado nulo (de índice `t2`) es el estado inicial, por lo que su índice es el valor devuelto por `expresion` y sus entradas `prox1` y `prox2` apuntan a los estados iniciales de las dos expresiones. Obsérvese que éstas se construyen en orden inverso al que se podría esperar, porque el valor de `índice` para el estado no operativo no se conoce hasta que se haya hecho la llamada recursiva de `expresion`.

La función para *<termino>* construye primero la máquina para un *<factor>* y luego, si es necesario, fusiona el estado final de esta máquina con el estado inicial de la máquina para otro *<termino>*. Esto es más fácil de hacer que decir, puesto que `índice` es el índice del estado final de la llamada a `factor`:

```

int termino()
{
    int t, r;
    r = factor();
    if (( p[j] == ')') || letra(p[j])) t = termino();
    return r;
}

```

Se ignora simplemente el índice del estado inicial devuelto por la llamada a `termino`: C++ exige ponerlo en alguna parte, por lo que se coloca en una variable temporal `t`.

La función para `<factor>` utiliza técnicas similares para manejar sus tres casos: un paréntesis implica una llamada recursiva a `expresion`; una `v` llama a la simple concatenación de un nuevo estado; y un `*` implica operaciones similares a las de `expresion`, de acuerdo con el diagrama de clausura que se vio en la sección anterior:

```

int factor()
{
    int t1, t2, r;
    t1 = estado;
    if (p[j] == '(')
    {
        j++; t2 = expresion();
        if (p[j] == ')') j++;
        else error();
    }
    else if (letra(p[j]))
    {
        actualiza_estado(estado, p[j], estado+1, estado+1);
        t2 = estado; j++; estado++;
    }
    else error();
    if (p[j] != '*') r = t2;
    else
    {
        actualiza_estado(estado, ' ', estado+1, t2);
        r = estado; prox1[t1-1] = estado;
        j++; estado++;
    }
    return r;
}

```

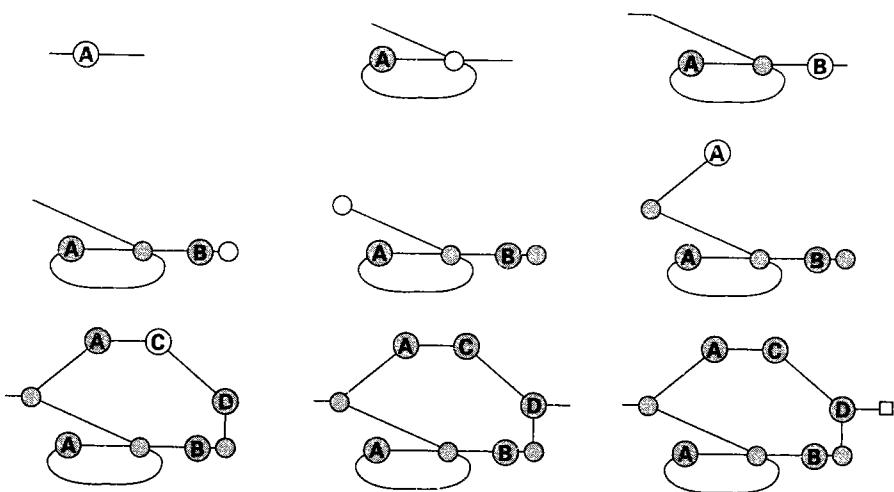


Figura 21.3 Construcción de una máquina de reconocimiento de patrones para $(A^*B+AC)D$.

La Figura 21.3 muestra cómo se construyen los estados para el patrón $(A^*B+AC)D$, del ejemplo del capítulo anterior. Primero se construye el estado 1 para la A. Luego se construye el estado 2 para el operando clausura y se agrega el estado 3 para la B. A continuación se encuentra el «+» y los estados 4 y 5 son construidos por expresion, pero no pueden llenarse sus campos hasta después de una llamada recursiva a expresion, lo que se traduce en la construcción de los estados 6 y 7. Por último, el estado 8 trata la concatenación de D, quedando el estado 9 como estado final.

El paso final del desarrollo de un algoritmo general de reconocimiento de patrones descritos por expresiones regulares consiste en poner estos procedimientos junto con el procedimiento concordar:

```
void concordar_todos(char *a)
{
    j = 0; estado = 1;
    prox1[0] = expresion();
    actualiza_estado(0, ' ', prox1[0], prox1[0]);
    actualiza_estado(estado, ' ', 0, 0);
    while (*a) cout << concordar(a++) << ' ';
    cout << '\n';
}
```

Este programa imprime, para la posición de cada carácter de una cadena de texto a, la longitud de la subcadena más corta que, comenzando en esa posición, concuerda con un patrón p (0 si no hay concordancia).

Compilador de compiladores

El programa que se ha desarrollado en este capítulo y en el anterior para el reconocimiento de patrones descritos por expresiones regulares es eficaz y bastante útil. Una versión ligeramente mejorada de este programa (capaz de manejar caracteres «sin importancia», etc.) tiene todas las posibilidades de encontrarse entre las herramientas más utilizadas en numerosos sistemas de información.

Es interesante (algunos pudieran decir confuso) reflexionar sobre este algoritmo desde un punto de vista más filosófico. En este capítulo se han utilizado analizadores sintácticos para explicar la estructura de las expresiones regulares, sobre la base de su descripción formal utilizando una gramática libre de contexto. Se ha utilizado una gramática libre de contexto para especificar un «patrón» particular: una serie de caracteres con paréntesis correctamente equilibrados. El analizador sintáctico comprueba si el patrón aparece en la cadena de entrada (pero considera que la concordancia es válida sólo si cubre la cadena completa de entrada). Así pues, analizar si una cadena de entrada pertenece al conjunto de cadenas definidas por una gramática libre de contexto, y hacer un reconocimiento de patrones para comprobar si la cadena de entrada pertenece al conjunto de cadenas definidas por una expresión regular, ¡es esencialmente la misma función! La diferencia fundamental es que las gramáticas libres de contexto son capaces de describir una clase mucho más amplia de cadenas. Por ejemplo, las expresiones regulares no pueden describir el conjunto de todas las expresiones regulares.

Otra diferencia en los programas es que la gramática libre de contexto está «integrada» en el analizador sintáctico, mientras que el procedimiento concordar está «dirigido por tablas»: el mismo programa funciona para toda expresión regular, una vez que se haya puesto en un formato apropiado. Parece posible construir analizadores sintácticos «dirigidos por tablas», de modo que el mismo programa pueda servir para analizar todos los lenguajes que se puedan describir con gramáticas libres de contexto. Un *generador de analizadores sintácticos* es un programa que recibe como entrada una gramática y produce como salida un analizador sintáctico para el lenguaje descrito por esa gramática. Esto se puede llevar más lejos: se pueden construir compiladores dirigidos por tablas en términos de los lenguajes de entrada y de salida. Un *compilador de compiladores* es un programa que recibe como entrada dos gramáticas (así como una especificación de las relaciones entre ellas) y produce, como salida, un compilador capaz de traducir las cadenas de uno de los lenguajes al otro.

Existen generadores de analizadores sintácticos y compiladores de compiladores para uso general en la mayoría de los entornos informáticos. Son herramientas muy útiles que se pueden utilizar con relativamente poco esfuerzo para producir analizadores sintácticos y compiladores eficaces y fiables. Por otra parte, los analizadores sintácticos recursivos descendentes del tipo considerado en este capítulo son bastante útiles para gramáticas simples como las que se encuentran

en muchas aplicaciones. Así pues, al igual que con muchos de los algoritmos que se han considerado, se dispone de un método directo apropiado para aplicaciones donde no se justifica un gran esfuerzo de implementación. Además, se dispone de varios métodos avanzados que permiten significativas mejoras del rendimiento en aplicaciones a gran escala. Pero, como ya se dijo, sólo se ha arañado la superficie de un campo que ha sido objeto de una extensa investigación.

Ejercicios

1. ¿Cómo encontraría un analizador sintáctico recursivo descendente un error en una expresión regular incompleta, como $(A+B)^*BC^+$?
2. Obtener el árbol de análisis sintáctico para la expresión regular $((A+B)+(C+D)^*)^*$.
3. Ampliar la gramática de las expresiones aritméticas para que incluya los operadores de exponentiación, división y módulo.
4. Obtener una gramática libre de contexto que describa todas las cadenas que no tienen más de dos 1 consecutivos.
5. ¿Cuántas llamadas a procedimientos se utilizan por el analizador sintáctico recursivo descendente para reconocer una expresión regular en términos del número de operaciones de concatenación, *or* y de clausura y del número de paréntesis?
6. Obtener los arrays *carac*, *prox1* y *prox2* que resultan al construir la máquina de reconocimiento de patrones para el patrón $((A+B)+(C+D)^*)^*$.
7. Modificar la gramática de las expresiones regulares para manejar la función *not* y los caracteres «sin importancia».
8. Construir un programa general de reconocimiento de patrones descritos por expresiones regulares basado en la gramática mejorada obtenida en la pregunta anterior.
9. Eliminar la recursión de un compilador recursivo descendente y simplificar el código obtenido tanto como sea posible. Comparar los tiempos de ejecución de ambos métodos (recursivo y no recursivo).
10. Escribir un compilador para las expresiones aritméticas simples descritas por la gramática del texto. Este compilador debe generar una lista de «instrucciones» para una máquina hipotética que sea capaz de ejecutar las siguientes operaciones: *poner* el valor de una variable en la pila; *sumar* los dos valores superiores de la pila, eliminándolos de ella y poniendo en su lugar el resultado; y *multiplicar* los dos valores superiores de la pila, de forma análoga.

Compresión de archivos

Los algoritmos que se han estudiado hasta ahora han sido diseñados, en su mayor parte, para que utilicen el menor *tiempo* posible, quedando la economía de *espacio* en un segundo plano. En esta sección se examinarán algunos algoritmos concebidos a la inversa, es decir, para utilizar el menor espacio posible sin consumir demasiado tiempo. Irónicamente, las técnicas a examinar para economizar espacio se basan en métodos de «codificación» que provienen de la teoría de la información que se desarrolló para disminuir el volumen de información necesaria en los sistemas de comunicación con la intención, en su origen, de ganar tiempo (no espacio).

En general, la mayor parte de los archivos tienen un gran nivel de redundancia. Los métodos que se examinarán reducen el espacio aprovechando el hecho de que muchos archivos tienen un «contenido de información» relativamente bajo. Las técnicas de *compresión de archivos* sirven a menudo para archivos de texto (en los que ciertos caracteres aparecen con mucha más frecuencia que otros), para archivos de «exploración» de imágenes codificadas (que presentan grandes zonas homogéneas) y para archivos de representación digital de sonido y de otras señales analógicas (que pueden presentar gran número de patrones repetidos).

En este capítulo se va a considerar un algoritmo elemental bastante útil para resolver este problema y también un método avanzado «óptimo». La cantidad de espacio que se gana con estos métodos varía según las características de los archivos. Ganancias del 20 al 50 % de espacio son típicas en archivos de texto, y es posible alcanzar entre un 50 y un 90 % en archivos binarios. Para ciertos tipos de archivos, como por ejemplo los constituidos por bits aleatorios, se puede ahorrar muy poco. De hecho, es interesante comprobar que cualquier método de compresión de propósito general puede aumentar el tamaño de algunos archivos (si no fuera así, sería posible, aplicando continuamente el método, obtener archivos arbitrariamente pequeños).

Por una parte, se puede argumentar que las técnicas de compresión de archivos son ahora menos importantes que lo que fueron hace tiempo porque el coste

de los dispositivos de almacenamiento ha caído drásticamente y un usuario medio puede tener a su alcance mayor capacidad de almacenamiento que la que tenía en el pasado. Pero, al contrario, también se puede argumentar que las técnicas de compresión de archivos son ahora más importantes que nunca, porque al poner en juego un gran volumen de almacenamiento, los ahorros que se pueden lograr son muy grandes. Las técnicas de compresión son también apropiadas para los dispositivos de almacenamiento que permiten accesos muy rápidos y que, por naturaleza, son relativamente caros (y por consiguiente pequeños).

Codificación por longitud de series

El tipo más simple de redundancia que se puede encontrar en un archivo son las largas series de caracteres repetidos. Por ejemplo, considérese la cadena siguiente:

AAAABBBAABBBBCCCCCCCABCBAABBBCCCD

Esta cadena se puede codificar de forma más compacta reemplazando cada repetición de caracteres por un solo ejemplar del carácter repetido seguido del número de veces que se repite. Sería mejor decir que esta cadena consiste en 4 letras A, seguidas de 3 B, seguidas de 2 A, seguidas de 5 B, etc. Esta forma de comprimir una cadena se denomina *codificación por longitud de series*. En el caso de largas series, los ahorros pueden ser espectaculares. Existen varias formas de realizar esta idea, dependiendo de las características de la aplicación. (¿Las series tienden a ser relativamente largas? ¿Cuántos bits se necesitan para codificar los caracteres?) A continuación se verá un método particular, para presentar después otras opciones.

Si se sabe que las cadenas contienen sólo letras, entonces es posible codificarlas introduciendo los dígitos entre las letras. La cadena anterior se podría codificar como:

4A3BAA5B8CDABCB3A4B3CD

Aquí «4A» significa «cuatro letras A», y así sucesivamente. Obsérvese que no merece la pena codificar las series de longitud uno o dos, puesto que para la codificación se necesitan dos caracteres.

En archivos binarios se utiliza una versión mejorada de este método con la que se obtienen grandes ahorros de espacio. La idea consiste simplemente en almacenar las longitudes de las series, aprovechando el hecho de que se componen de 0 o 1, para evitar almacenar estos caracteres. Esto supone que hay pocas series cortas (sólo se gana espacio si la longitud de la serie es mayor que el número de bits que se necesitan para representar dicha longitud en binario), pero

0000000000000000000000000000000011111111111110000000000	28	14	9		
00000000000000000000000000000011111111111111000000000	26	18	7		
000000000000000000000000000000111111111111111000000000	23	24	4		
000000000000000000000000000000111111111111111111111000	22	26	3		
000000000000000000000000000000111111111111111111111110	20	30	1		
000000000000000000000000000000111111110000000000000000001111111	19	7	18	7	
00000000000000000000000000000011111000000000000000000000011111	19	5	22	5	
0000000000000000000000000000001110000000000000000000000000111	19	3	26	3	
0000000000000000000000000000001110000000000000000000000000111	19	3	26	3	
0000000000000000000000000000001110000000000000000000000000111	19	3	26	3	
0000000000000000000000000000001110000000000000000000000000111	19	3	26	3	
00000000000000000000000000000011110000000000000000000000001110	20	4	24	3	1
0000000000000000000000000000001110000000000000000000000000111000	22	3	20	3	3
011	1	50			
011	1	50			
011	1	50			
011	1	50			
011	1	50			
0110011	1	2	46	2	

Figura 22.1 Una matriz de puntos típica, con información de la codificación por longitud de series.

ningún método de longitud de series es eficaz a menos que la mayor parte de las series sean largas.

La Figura 22.1 es una representación «por pixels» de la letra «q (apaizada)». Esta figura es representativa del tipo de información que se puede procesar por un sistema de formateo de texto (como el que se ha utilizado para imprimir este libro). A la derecha figura una lista de los números que se podrían utilizar para almacenar la letra en forma comprimida. Así, la primera línea consiste en 28 «0» seguidos de 14 «1», seguidos de otros 9 «0» más, etc. Las 63 informaciones de esta tabla más el número de bits por línea (51) contienen suficiente información para reconstruir el array de bits (en particular destaca que no se necesita ningún carácter de «fin de línea»). Si se utilizan 6 bits para representar cada longitud, entonces el archivo completo se representa con 384 bits, un ahorro sustancial comparado con los 975 bits que se necesitan para almacenarlo en forma explícita.

La codificación por longitud de series necesita representaciones separadas para los elementos del archivo y los de su versión codificada, por lo que no puede aplicarse en todos los archivos. Esto puede ser un inconveniente: por ejemplo, el método de compresión de archivos de caracteres sugerido anteriormente no funciona con cadenas que contienen dígitos. Si se utilizan otros caracteres para codificar las longitudes de las series, no podría aplicarse el método a las cadenas que contengan esos caracteres. Para ilustrar una forma de codificar cualquier cadena escrita por medio de un alfabeto fijo de caracteres, utilizando sólo ca-

racteres de ese alfabeto, supóngase que se dispone sólo de las 26 letras del alfabeto (y de espacios en blanco).

¿Cómo se puede lograr que algunas letras representen dígitos y otras formen parte de la cadena que se va a codificar? Una solución consiste en utilizar un carácter con pocas probabilidades de aparecer en el texto, al que se denomina *carácter de escape*. Cada aparición de dicho carácter indica que las dos letras siguientes forman un par (longitud, carácter), en el que la i -ésima letra del alfabeto representa una longitud igual a i . De esta manera, tomando Q como carácter de escape, la cadena del ejemplo se representaría por:

QDABBAAQEBQHCDABCAAAQDBCCCD

La combinación del carácter de escape, de la longitud de la serie y de una copia del carácter repetido se denomina *secuencia de escape*. Obsérvese que no merece la pena codificar series de menos de cuatro caracteres, ya que se necesitan al menos tres para codificar cualquier serie.

Pero ¿qué pasa si el carácter de escape aparece también en la serie de entrada? No se puede ignorar esta posibilidad, porque es difícil asegurar que no puede aparecer un carácter en particular. (Por ejemplo, alguien puede tratar de codificar una cadena que ya se ha codificado.) Una solución a este problema consiste en utilizar para representar al carácter de escape una secuencia de escape con una longitud de serie cero. De esta manera, en el ejemplo, el carácter espacio en blanco podría representar cero, y la secuencia de escape «Q<espacio» representaría a cualquier aparición de Q en la entrada. Es interesante notar que los únicos archivos que se «alargan» por este método de compresión son aquellos que contienen a Q. Si un archivo que ya ha sido comprimido se comprime otra vez, aumenta su longitud en un número de caracteres igual al número de secuencias de escape utilizadas.

Las series muy largas se pueden codificar con múltiples secuencias de escape. Por ejemplo, una serie de 51 A debería codificarse como QZAQYA, de acuerdo con las convenciones anteriores. Si se espera encontrar series muy largas, merecería la pena utilizar más de un carácter para codificar las longitudes.

En la práctica es aconsejable hacer que los programas de compresión y de descompresión sean algo más sensibles a los errores. Esto se puede lograr incluyendo una ligera redundancia en el archivo comprimido, de modo que el programa de descompresión pueda tolerar cualquier pequeño cambio accidental sufrido por el archivo entre la compresión y la descompresión. Por ejemplo, probablemente merece la pena insertar caracteres de «fin de línea» en la versión comprimida de letra «q» que se vio con anterioridad, de modo que el programa de descompresión pueda el mismo resincronizarse en caso de error.

La codificación por longitud de series no es particularmente eficaz en archivos de texto donde posiblemente el único carácter que tenga series repetidas es el espacio en blanco, ya que existen métodos más simples para codificar series de espacios en blanco repetidos. (Este método se utilizó con provecho en el pasado para comprimir archivos de texto creados a partir de lecturas de tarjetas

perforadas, que contenían necesariamente muchos espacios en blanco.) En los sistemas modernos nunca entran ni se almacenan series repetidas de espacios en blanco: las que aparecen al principio de una línea se codifican como «tabulaciones», y las que existen al final de las líneas se pueden ignorar utilizando los indicadores de «fin de línea». Una implementación de la codificación por longitud de series como la anterior (modificada para aceptar todos los caracteres representables) economiza solamente alrededor de un 4 % cuando se utiliza en un archivo de texto como el de este capítulo (y toda la economía proviene del ejemplo de la letra «q»!).

Codificación de longitud variable

En esta sección se examinará una técnica de compresión que permite ganar una cantidad considerable de espacio en archivos de texto (y en muchos otros tipos de archivos). La idea es abandonar la forma como se almacenan habitualmente los archivos de texto: en lugar de emplear siete u ocho bits por carácter, se utilizarán solamente unos pocos bits para los caracteres más frecuentes y algunos más para los que aparecen más raramente.

Será conveniente examinar, en un pequeño ejemplo, cómo se utiliza el código antes de considerar cómo se creó. Suponiendo que se desea codificar la cadena «ABRACADABRA», su codificación en el código binario compacto estándar, en el que la representación con cinco bits de i reproduce a la i -ésima letra del alfabeto (0 para los espacios en blanco), proporcionaría la siguiente serie de bits:

0000100010100100000100011000010010000001000101001000001

Para «decodificar» este mensaje se leen grupos de cinco bits y se convierten de acuerdo con la codificación binaria anterior. En este código estándar, la D, que aparece sólo una vez, necesita el mismo número de bits que la A, que aparece cinco veces. Con un código de longitud variable se puede alcanzar ahorros de espacio codificando los caracteres más frecuentemente utilizados con el menor número de bits posible, de forma que se minimice el número total de bits.

Se podría tratar de asignar la cadena más corta de bits a las letras más frecuentemente utilizadas, codificando A por 0, B por 1, R por 01, C por 10 y D por 11, y así ABRACADABRA se codificaría como

0 1 01 0 10 0 11 0 1 01 0

Esta cadena utiliza sólo 15 bits en lugar de los 55 anteriores, pero esto no es realmente un código porque depende de los «espacios en blanco» para delimitar los caracteres. Sin ellos, la cadena 010101001101010 se podría decodificar como RRRARBRRRA o como otras diferentes cadenas. A pesar de todo, 15 bits más

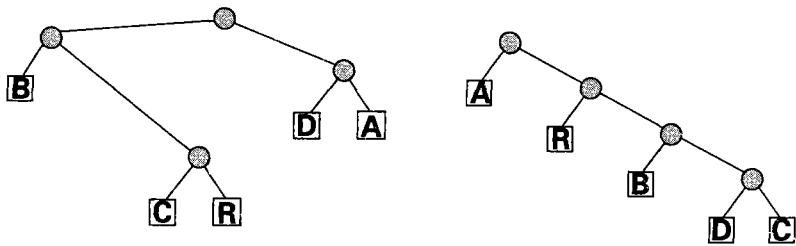


Figura 22.2 Dos tries de codificación para A, B, C, D y R.

10 delimitadores forman un código mucho más compacto que el código estándar, principalmente porque *no* se utilizan bits para codificar letras que no aparecen en el mensaje. Para ser objetivos, es preciso incluir también los bits del propio código, puesto que el mensaje no puede decodificarse sin él, y el código depende del mensaje (otros mensajes tendrán diferentes frecuencias de aparición de las letras). Más adelante se volverá a considerar este aspecto: por el momento solamente interesa ver hasta qué punto se puede comprimir el mensaje.

Los delimitadores no son necesarios si el código de un carácter no es el prefijo de otro. Por ejemplo, si se codifica A por 11, B por 00, C por 010, D por 10 y R por 011, no hay más que una sola forma de decodificar la cadena de 25 bits

1100011110101110110001111

Una forma fácil de representar el código es con un trie (ver Capítulo 17). En efecto, cualquier trie con M nodos externos se puede utilizar para codificar cualquier mensaje con M caracteres diferentes. Por ejemplo, la Figura 22.2 muestra dos códigos que se podrían utilizar para ABRACADABRA. El código de cada carácter se determina por el camino desde la raíz al carácter, con 0 para «ir a la izquierda» y 1 para «ir a la derecha», como es habitual en un trie. Así pues, el trie de la izquierda corresponde al código anterior; el trie de la derecha corresponde con el código que genera la cadena

01101001111011100110100

que es dos bits más corta. La representación por trie garantiza que el código de ningún carácter es el prefijo de otro, de modo que la cadena es únicamente decodificable a partir del trie. Comenzando en la raíz, se desciende por el trie de acuerdo con los bits del mensaje: cada vez que se encuentre un nodo externo, se da salida al carácter del nodo y se comienza de nuevo en la raíz.

Pero ¿qué trie es el mejor para utilizar? Existe una forma elegante de construir un trie que proporcione una cadena de bits de longitud mínima para cualquier mensaje. El método general para encontrar el código fue descubierto por D. Huffman en 1952 y se denomina *código de Huffman*. (La implementación que se verá utiliza algunas metodologías algorítmicas más modernas.)

	A	B	C	D	E	F	I	L	M	N	O	R	S	T	U	
k	0	1	2	3	4	5	6	9	12	13	14	15	18	19	20	21
cuenta[k]	11	6	1	5	3	4	1	6	2	3	7	4	2	2	1	3

Figura 22.3 Cuentas diferentes de cero para UNA CADENA SENCILLA A...

Construcción del código de Huffman

El primer paso en la construcción del código de Huffman es contar el número de veces que aparece (frecuencia de aparición) cada carácter en el mensaje que se va a codificar. Las siguientes instrucciones permiten llenar un array `cuenta[26]` con la cuenta de las frecuencias de aparición de un mensaje en una cadena `a`. (Este programa utiliza el procedimiento `indice` descrito en el Capítulo 19 para almacenar la cuenta de frecuencias de la i -ésima letra del alfabeto en `cuenta[i]`, utilizando `cuenta[0]` para los espacios en blanco.)

```
for (i=0; i<=26; i++) cuenta[i] = 0;
for (i=0; i < M; i++) cuenta[indice(a[i])]++;
```

Por ejemplo, supóngase que se desea codificar la cadena «UNA CADENA SENCILLA A CODIFICAR CON UN NÚMERO MÍNIMO DE BITS». La tabla de cuentas que se obtiene se muestra en la Figura 22.3: hay once espacios en blanco, seis A, una B, etcétera.

El siguiente paso es construir el trie de codificación de abajo hacia arriba de acuerdo con las frecuencias. Al construir el trie, se considerará como un árbol binario con las frecuencias almacenadas en los nodos: después de su construcción se considerará como un trie de codificación, al igual que antes. Primero se crea un nodo del árbol para cada frecuencia distinta de cero, como se muestra en el primer diagrama en la parte superior izquierda de la Figura 22.4 (el orden en el que aparecen los nodos se determina por la dinámica del algoritmo descrito a continuación, pero no es relevante para esta presentación). A continuación se toman los dos nodos con las frecuencias más pequeñas y se crea un nuevo nodo con estos dos como hijos y con un valor de frecuencia igual a la suma de los valores de los hijos, como se muestra en el segundo diagrama de la Figura 22.4. (Si existen más de dos nodos con el mismo valor mínimo de frecuencia se eligen dos cualesquiera.) Luego se busca entre los nodos restantes los dos con menor frecuencia y se vuelve a crear un nuevo nodo de la misma forma, tal y como se muestra en el tercer diagrama de la Figura 22.4. Continuando de esta manera, se van construyendo subárboles cada vez más grandes, a la vez que se reduce en cada paso el número de subárboles del bosque (se eliminan dos y se añade uno). Finalmente, todos los nodos se combinan en un solo árbol.

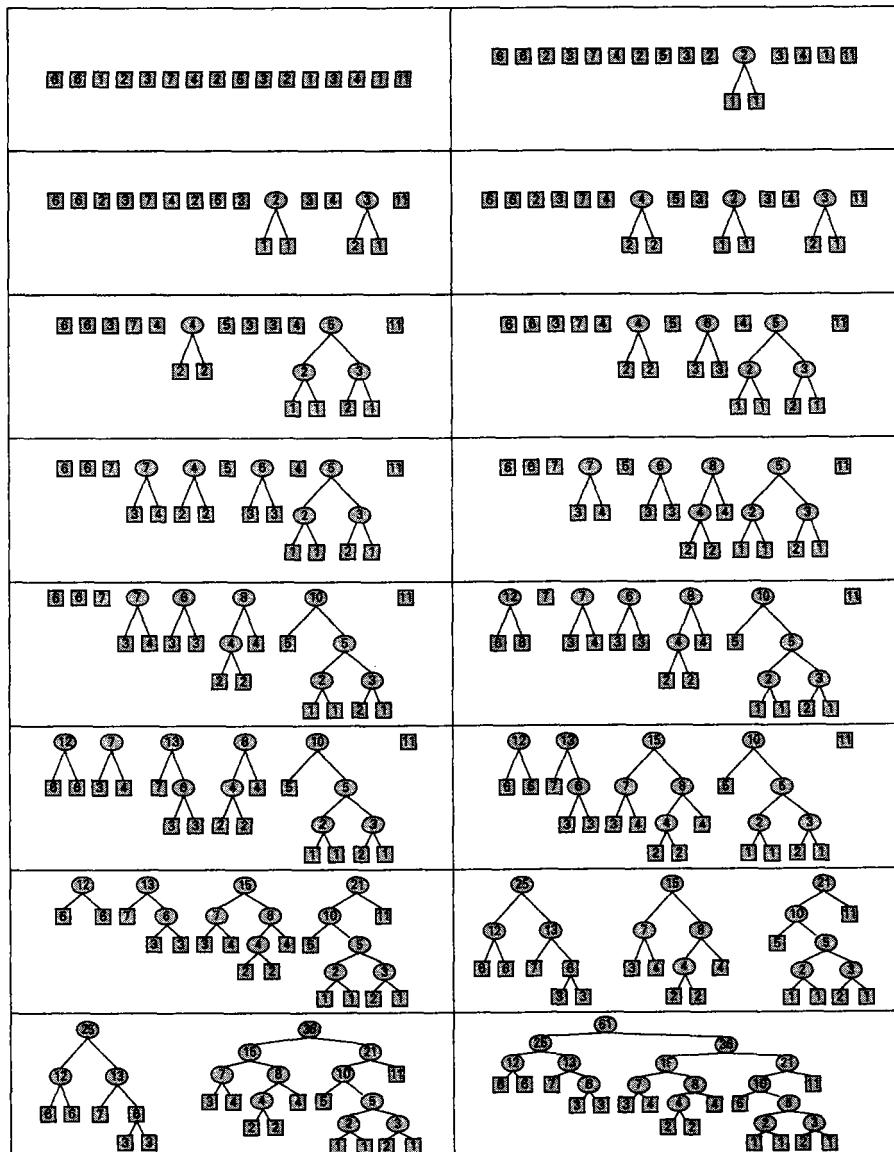


Figura 22.4 Construcción de un árbol de Huffman.

Obsérvese que los nodos con las frecuencias más bajas se encuentran bastante abajo en el árbol, y los nodos con frecuencias altas se sitúan cerca de la raíz. El número de la etiqueta de los nodos externos (cuadrados) de este árbol es la cuenta de la frecuencia, mientras que el número de cada nodo interno (redondos) es la suma de las etiquetas de sus dos hijos.

El código de Huffman se obtiene ahora fácilmente sustituyendo las frecuencias de los nodos del fondo por los caracteres asociados y considerando al árbol como un trie de codificación: cada bifurcación hacia la «izquierda» corresponde al bit de código 0 y hacia la «derecha» al de código 1.

Evidentemente, las letras con frecuencias altas están cerca de la raíz del árbol y se codifican con pocos bits, por lo que éste es un buen código; pero, ¿por qué es el mejor?

Propiedad 22.1 *La longitud del mensaje codificado es igual a la longitud ponderada del camino externo del árbol de frecuencias de Huffman.*

La «longitud ponderada del camino externo» de un árbol es la suma, para todos los nodos externos, del producto del «peso» (la cuenta de frecuencias) por la distancia a la raíz. Evidentemente ésta es una forma de calcular la longitud del mensaje: es equivalente a la suma, para todos los caracteres, del producto del número de apariciones de cada carácter por el número de bits asociado con cada aparición.■

Propiedad 22.2 *Ningún árbol con las mismas frecuencias en los nodos externos puede tener una longitud ponderada del camino externo inferior a la del árbol de Huffman.*

Cualquier árbol se puede reconstruir con el mismo procedimiento que se utilizó para construir el árbol de Huffman, pero no seleccionando necesariamente en cada paso los dos nodos de menor peso. Se puede demostrar por inducción que no hay mejor estrategia que la de tomar primero los dos pesos menores.■

Siempre que se escoja un nodo, puede ocurrir que haya varios nodos con el mismo peso. El método de Huffman no especifica lo que hay que hacer en este caso. Las diferentes opciones conducirán a códigos diferentes, pero todos ellos codificarán el mensaje con el mismo número de bits. Por ejemplo, en la Figura 22.5 se muestra otro trie para el ejemplo. El código de A es 000, el de B 110110, el de C 1100, etc. Este árbol es estructuralmente algo diferente del construido en la Figura 22.4. Puede que el lector quiera comprobar que tienen la misma longitud ponderada del camino externo. (El número más pequeño encima de cada nodo del árbol es un índice que sirve como referencia al tratar la implementación que se proporciona más adelante.)

La descripción anterior muestra un esbozo general de la construcción de un código de Huffman en función de las operaciones algorítmicas que se han estudiado. Como es habitual, el paso de esta descripción a una implementación

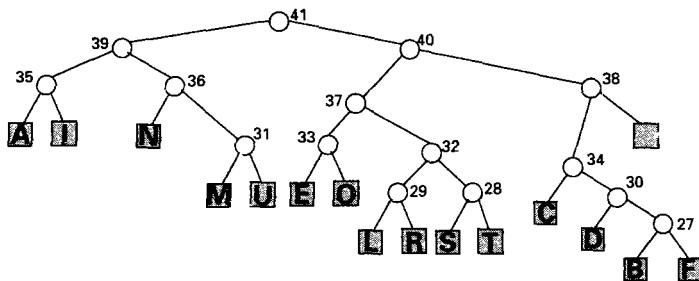


Figura 22.5 Un trie del código de Huffman para UNA CADENA SENCILLA...

concreta es bastante instructivo, por lo que a continuación se examinarán los detalles de la implementación.

Implementación

La construcción del árbol de frecuencias implica un proceso general de extraer el elemento más pequeño de un conjunto de elementos desordenados, por lo que se utiliza la clase CP del Capítulo 11 para construir y mantener una cola de prioridad de los valores de las frecuencias. La utilización de esta cola para construir el árbol descrito anteriormente es directa:

```

for (i = 0; i <= 26; i++)
    if (cuenta[i]) cp.insertar(cuenta[i], i);
for (; !cp.vacia(); i++)
{
    t1 = cp.suprimir(); t2 = cp.suprimir();
    padre[i]=0; padre[t1]=i;padre[t2]=-i;
    cuenta[i]= cuenta[t1] + cuenta[t2];
    if (!cp.vacia()) cp.insertar(cuenta[i], i);
}

```

Primero, se insertan en la cola de prioridad las cuentas diferentes de cero. Luego se extraen los dos elementos más pequeños, se suman sus frecuencias y se inserta el resultado en la cola, continuando con el mismo procedimiento hasta vaciar la cola. En cada paso se crea una nueva cuenta y se disminuye el tamaño de la cola. Este proceso crea $N-1$ nuevas cuentas, una para cada uno de los nodos internos que se están creando en el árbol. El índice i continúa refiriéndose al array cuenta, por lo que el primer nodo interno tiene índice 27, etc. Se «crean» nuevos nodos internos a través de $i++$. El propio árbol está represen-

k	0	1	2	3	4	5	6	9	12	13	14	15	18	19	20	21
cuenta[k]	11	6	1	5	3	4	1	6	2	3	7	4	2	2	1	3
padre[k]	-38	35	27	34	30	33	-27	35	29	31	36	-33	-29	28	-28	-31
k	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	
cuenta[k]	2	3	4	5	6	7	8	10	12	13	15	21	25	36	61	
padre[k]	-30	-32	32	-34	-36	-37	37	38	39	-39	40	-40	41	-41	0	

Figura 22.6 Representación de enlaces del trie de Huffman de la Figura 22.5.

tado por un array de enlaces «padres»: `padre[t]` es el índice del padre del nodo cuyo peso está en `cuenta[t]`. El signo de `padre[t]` indica si el nodo es un hijo derecho o izquierdo del padre. La Figura 22.6 muestra el contenido completo de las estructuras de datos del árbol de la Figura 22.5. Por ejemplo, se tiene `cuenta[40]=36`, `padre[40]=-41` y `cuenta[41]=61` (lo que indica que el nodo de peso 36 tiene índice 40 y es el hijo derecho de un parente que tiene índice 41 y peso 61).

El trie es suficiente para describir el propio código; para aclarar este punto a continuación se considerará cómo construir explícitamente el código. La Figura 22.7 muestra el código completo del ejemplo representado por dos arrays: los `long[k]` bits más a la derecha en la representación binaria del entero `codigo[k]` forman el código de la k-ésima letra. Por ejemplo, I es la novena letra y tiene como código 001, por lo que `codigo[9]=1`, y `long[9]=3`, lo que indica que el código es los tres bits más a la derecha de la representación binaria del número 1, o sea 001. El siguiente segmento de programa convierte a la representación trie del código (el array `padre`, que se muestra en la Figura 22.6) en el propio código de Huffman.

```

for ( k = 0; k <= 26; k++)
{
    i = 0; x = 0; j = 1;
    if (cuenta[k])
        for (t=padre[k]; t; t=padre[t], j+=j, i++)
            if (t < 0) { x +=j; t = -t; }
        codigo[k] = x; long[k] = i;
}

```

Los arrays `codigo` y `long` se calculan de forma directa utilizando el array `padre` para ascender por el árbol.

El mensaje podría codificarse recorriendo el trie de esta forma para cada carácter del mensaje. Como alternativa, se pueden utilizar directamente los arrays `codigo` y `long` para codificar el mensaje:

```

for ( j = 0; j < M; j++)
    for ( i = long[indice(a[j])]; i } 0; i--)
        cout << ((codigo[indice(a[j])] >> i-1) & 1);

```

El mensaje del ejemplo se codifica con sólo 228 bits en lugar de los 300 que se utilizarían en la codificación directa, lo que supone un 24 % de ahorro:

```

011101000011110000011010100001000011101101000010110000110
100101000001110001111001001110100011101110011000001010111
11100100101011101110101110100111011010001010110011110110001
0100010110100111110101000111101100011011110110111

```

Ahora, como ya se mencionó, se debe almacenar el árbol o bien enviarlo junto con el mensaje para decodificarlo. Afortunadamente, esto no presenta ninguna dificultad real. No se necesita más que almacenar el array código, porque el trie de búsqueda por residuos que resulta de insertar las entradas del array en un árbol inicialmente vacío es el árbol de decodificación.

	A	B	C	D	E	F	I	L	M	N	O	R	S	T	U	
k	0	1	2	3	4	5	6	9	12	13	14	15	18	19	20	21
codigo[k]	7	0	54	12	26	8	55	3	20	6	2	9	21	22	23	7
long[k]	3	3	6	4	5	4	6	3	5	4	3	4	5	5	5	4

```

111 000 110110 1100 11010 1000 110111 001 10100 0110 010 1001 10101 10110 10111 0111

```

Figura 22.7 Código de Huffman para UNA CADENA SENCILLA A...

Así, el ahorro de espacio antes mencionado no es totalmente exacto, porque el mensaje no se puede decodificar sin el trie y se debe tener en cuenta el coste que significa almacenar el árbol (esto es, el array código) junto con el mensaje. Por lo tanto, la codificación de Huffman sólo es efectiva para archivos largos donde el ahorro de espacio en el mensaje es suficiente como para compensar el coste, o en situaciones donde el trie de codificación puede calcularse previamente y reutilizarse para un gran número de mensajes. Por ejemplo, un trie basado en las frecuencias de aparición de las letras de un idioma determinado podría utilizarse en documentos de texto. De la misma forma, un trie basado en las frecuencias de aparición de caracteres en programas en C++ se podría utilizar para la codificación de programas (por ejemplo, «» estaría cerca de la raíz de un tal trie). Un algoritmo de codificación de Huffman permite unas ganancias de espacio de alrededor de un 23% en el texto de este capítulo.

Como siempre, para archivos verdaderamente aleatorios, incluso este inge-

nioso esquema de codificación no funcionará bien porque cada carácter aparecerá aproximadamente el mismo número de veces, lo que conduce a un árbol de codificación completamente equilibrado y a un número igual de bits por letra del código.

Ejercicios

1. Implementar los procedimientos de compresión y descompresión descritos en el texto para el método de codificación por longitud de series, considerando un alfabeto fijo y utilizando la letra Q como carácter de escape.
2. ¿Podría aparecer la serie «QQ» en un archivo comprimido por el método descrito en el texto? ¿Podría aparecer «QQQ»?
3. Implementar los procedimientos de compresión y descompresión descritos en el texto para el método de codificación de archivos binarios.
4. El dibujo de la letra «q» del texto se puede procesar como una serie de caracteres de cinco bits. Analizar las ventajas e inconvenientes de recurrir a esta técnica en el método de codificación por longitud de series basado en caracteres.
5. Mostrar el proceso de construcción del árbol de codificación de Huffman cuando se aplica el método de este capítulo a la cadena «ABRACADABRA». ¿Cuántos bits necesita el mensaje codificado?
6. ¿Cuál es el código de Huffman de un archivo binario? Dar un ejemplo que muestre el máximo número de bits que se podría utilizar en un código de Huffman de un archivo ternario (tres valores) de N caracteres.
7. Suponiendo que las frecuencias de aparición de todos los caracteres a codificar son diferentes. ¿Es único el árbol de codificación de Huffman?
8. La codificación de Huffman podría generalizarse de forma directa para codificar en caracteres de dos bits (utilizando árboles de 4 vías). ¿Cuáles serían las principales ventaja e inconveniente de una técnica como ésta?
9. ¿Cuál sería el resultado de dividir una cadena codificada por Huffman en caracteres de cinco bits y aplicar el código de Huffman a esa nueva cadena?
10. Implementar un procedimiento para *decodificar* una cadena codificada por Huffman, conociendo los arrays *código* y *long*.

Criptología

En el capítulo anterior se vieron métodos para codificar cadenas de caracteres con objeto de ahorrar espacio. Por supuesto, existe otra razón muy importante para codificar cadenas de caracteres: mantenerlas secretas.

La criptología, el estudio de los sistemas de comunicaciones secretas, está constituida por dos campos de estudio complementarios: la *criptografía*, o el diseño de sistemas de comunicaciones secretas, y el *criptoanálisis*, o el estudio de las formas de transgredir los sistemas de comunicaciones secretas. La criptología se aplicó inicialmente a los sistemas de comunicaciones militares y diplomáticos, pero en la actualidad están apareciendo otras aplicaciones importantes. Dos de los principales ejemplos son los sistemas de administración de archivos de las computadoras (en los que cada usuario desea que sus archivos se mantengan como privados) y los sistemas de transferencia electrónica de fondos (en los que se tratan grandes cantidades de dinero). Un usuario de computadora desea mantener sus archivos tan secretos como lo están sus papeles en su archivador y un banco desea que las transferencias electrónicas de fondos sean tan seguras como las que se hacen en un coche blindado.

Excepto en las aplicaciones militares, se supone que los criptógrafos son los «chicos buenos» y los criptoanalistas los «chicos malos»: el objetivo es proteger los archivos informáticos y las cuentas de un banco de los ladrones. Si este punto de vista parece poco amistoso, se debe recordar (sin tratar de filosofar mucho) que al utilizar la criptografía se supone ¡la existencia de la enemistad! Por supuesto, los «chicos buenos» deben saber algo de criptoanálisis, puesto que la mejor forma de saber si un sistema es seguro es tratar de violarlo uno mismo. (Además hay muchos ejemplos documentados sobre guerras que han finalizado, salvándose así muchas vidas, gracias a éxitos del criptoanálisis.)

La criptología tiene muchos lazos con la informática y los algoritmos, especialmente con los algoritmos aritméticos y de procesamiento de cadenas que se acaban de estudiar. En efecto, esta relación entre el arte (¿la ciencia?) de la criptología con las computadoras y la informática está ahora empezando a comprenderse. Al igual que los algoritmos, los sistemas de cripto aparecieron mu-

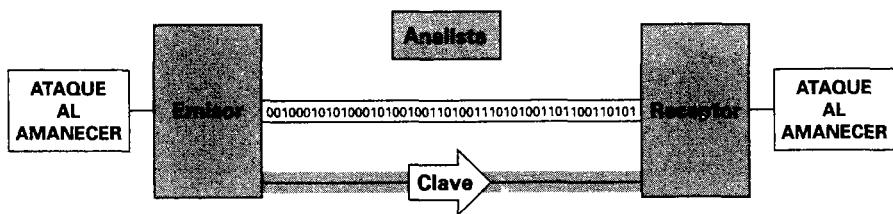


Figura 23.1 Un sistema de cripto típico.

cho antes que las computadoras. El diseño de sistemas secretos y el de los algoritmos tienen una herencia común, y la misma gente se siente atraída por ambos.

No es fácil decir qué rama de la criptología ha sido la más afectada por la aparición de las computadoras. Los criptógrafos tienen ahora a su disposición máquinas mucho más poderosas, pero también es más fácil que cometan errores. Los criptoanalistas cuentan con herramientas mucho más eficaces para «romper» los códigos, pero éstos son ahora mucho más complejos. El criptoanálisis puede suponer una gran carga de trabajo para las máquinas; no sólo fue una de las primeras áreas de aplicación de las computadoras, sino que se mantiene como uno de los dominios principales de aplicación de las modernas supercomputadoras.

Más recientemente, la amplia difusión de las computadoras ha generado la aparición de una gran variedad de nuevas aplicaciones importantes de la criptología, como se mencionó anteriormente. Se han desarrollado nuevos métodos de criptografía para responder a las necesidades de tales aplicaciones, y esto ha llevado al descubrimiento de una relación fundamental entre la criptografía y un área importante de la teoría informática que se examinará brevemente en el Capítulo 45.

En este capítulo se verán algunas de las características básicas de los algoritmos criptográficos. No se entrará en el detalle de las implantaciones: la criptografía es realmente un campo para confíarselo a los expertos. Mientras que no es difícil «protegerse» cifrando la información con un sencillo algoritmo, es peligroso confiar en un método implantado por un profano.

Reglas del juego

El conjunto de elementos que permiten la comunicación segura entre dos personas se denomina colectivamente un *sistema de cripto*. La Figura 23.1 muestra la estructura canónica de un sistema de cripto típico.

El *emisor* envía un mensaje (denominado el *texto en claro*) al *receptor*, transformándolo en una forma secreta propicia para la transmisión (denominada

el *texto cifrado*) por medio de un algoritmo de criptografía (el *método de cifrado*) y algunos parámetros (*claves*). Para leer el mensaje, el receptor debe tener un algoritmo criptográfico equivalente (el *método de descifrado*) y los mismos parámetros clave que transformarán el texto cifrado en el texto original. Habitualmente se supone que el texto cifrado se envía por líneas de comunicación inseguras y que puede estar al alcance del criptoanalista. También se supone que el método de cifrado y el de descifrado son conocidos por el criptoanalista: su objetivo es recuperar el texto en claro a partir del texto cifrado, pero sin conocer las claves. Es de destacar que todo el sistema depende de algún método preliminar de comunicación entre el emisor y el receptor para ponerse de acuerdo sobre los parámetros claves. Por regla general, cuantas más claves haya, más seguro será el sistema, pero más incómodo de utilizar. Esta situación es análoga a la de los sistemas de seguridad más convencionales: la combinación de una caja fuerte es más segura cuantos más números tenga, pero es más difícil de recordar. La analogía con los sistemas convencionales también sirve para recordar que cualquier sistema de seguridad es tan fiable como lo sean las personas que tengan la clave.

Es importante recordar que las cuestiones económicas representan un papel importante en los sistemas de cripto. Serán razones económicas las que lleven a construir dispositivos de cifrado y descifrado simples (porque puede ser que se necesiten muchos y los dispositivos complicados cuestan más), y también habrá una motivación económica para reducir el número de informaciones claves a distribuir (porque pueden necesitar un método de comunicación muy seguro y, por ello, caro). En el equilibrio entre el coste de implantación de un sistema criptográfico seguro y el coste de distribución de las informaciones claves, se encuentra el precio que los criptoanalistas están dispuestos a pagar para romper el sistema. En la mayoría de las aplicaciones, el objetivo del criptógrafo es desarrollar sistemas de bajo coste con la característica de que el criptoanalista debe invertir para leer los mensajes mucho más de lo que está dispuesto a pagar. En un pequeño número de aplicaciones, puede que se necesite un sistema de cripto «ciertamente seguro», que pueda garantizar que el criptoanalista nunca podrá leer los mensajes, sin importar lo que esté dispuesto a pagar por ello. (Los gastos muy altos en ciertas aplicaciones de criptología implican naturalmente que se invierten grandes cantidades de dinero para el criptoanálisis.) En el diseño de algoritmos se intenta seguir la pista a los costes para seleccionar el mejor algoritmo; en criptología, los costes desempeñan un papel fundamental en el proceso de diseño.

Métodos elementales

Entre los métodos de cifrado más simples (y de los más antiguos) se encuentra la *cifra de César*: si una letra del texto en claro es la N -ésima del alfabeto se

reemplaza por la $(N + K)$ -ésima letra del alfabeto, siendo K un cierto entero fijo (*César* utilizaba $K = 3$). La siguiente tabla muestra un mensaje cifrado utilizando este método con $K = 1$:

Texto en claro: A T A Q U E A L A M A N E C E R
 Texto cifrado: B U B R V F A B M A B N B O F D F S

El método es débil porque el criptoanalista sólo tiene que adivinar el valor de K : intentando con cada una de las 26 opciones, podrá estar seguro de leer el mensaje.

Un método mucho mejor consiste en utilizar una tabla general para definir la sustitución a efectuar: para cada letra del texto en claro la tabla dice qué letra poner en el texto cifrado. Por ejemplo, si la tabla ofrece la correspondencia

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	W	X	Y	Z	
N	U	E	V	O	S	B	R	I	L	Y	H	A	M	T	W	Z	Y	G	Q	P	F	J	V	K	C

entonces el mensaje quedará cifrado de la siguiente forma:

Texto en claro: A T A Q U E A L A M A N E C E R
 Texto cifrado: U Q U Z P S N U H N U A U M S V S Y

Esta técnica es mucho más poderosa que la simple cifra de *César* porque el criptoanalista tendría que ensayar con muchas tablas (alrededor de $27! > 10^{28}$) para estar seguro de leer el mensaje. Sin embargo, las cifras basadas en sustituciones simples como éstas son fáciles de romper debido a las frecuencias de letras inherentes a un lenguaje. Por ejemplo, puesto que A es la letra más frecuente en los textos en español, el criptoanalista ya tendría parte del trabajo hecho si comenzara buscando en el texto cifrado cuál es la letra más frecuente y la reemplazara por A. Aunque ésta puede no ser la selección correcta, con seguridad es mejor que ensayar las 26 letras ciegamente. La situación se hace más fácil (para el criptoanalista) cuando se tienen en cuenta combinaciones de dos letras: ciertas combinaciones (tal como QJ) nunca aparecen en un texto en español mientras que otras (como LA) son frecuentes. Examinando las frecuencias de las letras y de las combinaciones de letras, un criptoanalista puede romper fácilmente un cifrado por sustitución.

Se puede complicarlo más utilizando varias tablas. Un ejemplo simple de esto es una extensión de la cifra de *César* denominada la *cifra de Vigenere*: se utiliza una pequeña clave repetida para determinar el valor de K para cada letra. En cada paso, el índice de la letra de la clave se añade al de la letra del texto en claro para determinar el índice de la letra del texto cifrado. El ejemplo de texto en claro, con la clave ABC queda cifrado de la siguiente forma:

Clave: A B C A B C A B C A B C A B C
 Texto en claro: A T A Q U E A L A M A N E C E R

Texto cifrado: B V D R W H A C O A C P B P H D G U

Por ejemplo, la última letra del texto cifrado es U, la vigésimo primera del alfabeto, porque la letra correspondiente en el texto en claro es R (la decimoctava letra) y la letra correspondiente en la clave es C (la tercera letra).

Evidentemente, la cifra de *Vigenere* se puede hacer más complicada utilizando tablas generales diferentes para cada letra del texto en claro (en lugar de simples desplazamientos). También es obvio que cuanto más larga es la clave más eficaz será la cifra. En efecto, si la clave es tan larga como el texto en claro se tiene la cifra de *Vernam*, comúnmente denominada *clave para una vez*. Éste es probablemente el único sistema de cripto seguro que se conoce, y se dice de él que se ha utilizado en la *línea directa* Washington-Moscú y en otras aplicaciones vitales. Puesto que cada letra clave se utiliza una sola vez, el criptoanalista no puede hacer nada mejor que ensayar todas las claves posibles para cada letra del mensaje, una situación desesperante, ya que esto es tan difícil como ensayar todos los mensajes posibles. Sin embargo, el utilizar cada letra clave una sola vez genera un problema de distribución de claves bastante serio, por lo que esta *clave para una vez* es útil para mensajes relativamente cortos que se deben enviar con poca frecuencia.

Si el mensaje y la clave están codificados en binario, una técnica más común de cifrado letra a letra consiste en utilizar la función «o-exclusivo»: para cifrar el texto en claro se aplica un «o-exclusivo» (bit a bit) con la clave. Una característica interesante de este método es que la operación de descifrar es la misma que la de cifrar: el texto cifrado es el o-exclusivo del texto en claro y de la clave, pero la aplicación de otro o-exclusivo al texto cifrado y a la clave lleva de nuevo al texto en claro. Se observa también que el o-exclusivo de los textos cifrado y en claro da la clave. Esto puede sorprender a primera vista, pero realmente muchos sistemas criptográficos tienen la propiedad de que el criptoanalista puede descubrir la clave si conoce el texto en claro.

Máquinas de cifrar/descifrar

Muchas aplicaciones criptográficas (por ejemplo, sistemas de voz en comunicaciones militares) implican la trasmisión de grandes volúmenes de datos, lo que hace imposible la utilización de la *clave para una vez*. Lo que se necesita es una aproximación a esta clave en la que se pueda generar un gran volumen de «pseudoclaves» a partir de una pequeña fracción, muy distribuida, de la verdadera clave.

Lo usual en tales situaciones es lo siguiente: el emisor alimenta una máquina de cifrar con algunas *variables de cifrado* (claves verdaderas), para generar una larga secuencia de bits de clave (pseudoclaves). El o-exclusivo de estos bits y del texto en claro forman el texto cifrado. El receptor, con una máquina similar y las mismas variables de cifrado, genera la misma secuencia de bits de

clave para aplicarle el o-exclusivo con el texto cifrado y recuperar el texto en claro.

En este contexto, la generación de claves es similar a la *dispersión* y a la generación de números aleatorios, por lo que los métodos de los Capítulos 16 y 35 son apropiados para la generación de claves. En efecto, algunos de los mecanismos presentados en el Capítulo 35 se desarrollaron en principio para utilizarlos en máquinas de cifrar/descifrar tales como las que se describen aquí. Sin embargo, los generadores de claves deben ser más complejos que los generadores de números aleatorios, porque existen técnicas para atacar a las máquinas simples. El problema es que el criptoanalista puede llegar fácilmente a obtener alguna parte del texto en claro (por ejemplo, los tiempos de silencio en un sistema de voz), y, por lo tanto, una parte de la clave. Si el criptoanalista dispone de suficiente información sobre la máquina, entonces lo que conozca de la clave le puede proporcionar suficientes pistas para permitir que en algún momento pueda deducir los valores de las variables de cifrado. Entonces puede simular el funcionamiento de la máquina y calcular todas las claves a partir de ese momento.

Los criptógrafos disponen de varias formas de evitar estos problemas. Una de ellas consiste en definir una parte de la arquitectura de la máquina en sí misma como una variable de cifrado. Por lo regular se supone que el criptoanalista conoce todo sobre la estructura de la máquina (quizás robaron alguna), excepto las variables de cifrado, pero si se utilizan algunas de éstas para «configurar» la máquina, entonces puede ser difícil encontrar sus valores. Otro método comúnmente utilizado para confundir al criptoanalista es la *cifra producto*, en el que se combinan dos máquinas diferentes para generar una compleja secuencia de claves (o para controlarse mutuamente). Otro método es la *sustitución no lineal*; aquí el paso del texto en claro al cifrado se hace por grandes segmentos, no bit a bit. El problema de estos métodos es que pueden ser demasiado complicados, incluso para el criptógrafo, y siempre puede existir la posibilidad de que se produzcan comportamientos degenerados para alguna selección de las criptovariables.

Sistemas de cripto de claves públicas

En aplicaciones comerciales tales como la transferencia electrónica de fondos y el (verdadero) correo electrónico, el *problema de la distribución de claves* es aún más crucial que en las aplicaciones tradicionales de la criptografía. La perspectiva de ofrecer a cada ciudadano grandes claves que deben cambiarse con frecuencia, en beneficio de la seguridad y la eficacia, inhibe ciertamente el desarrollo de tales sistemas. Sin embargo, en fechas recientes se han desarrollado métodos que prometen eliminar por completo el problema de la distribución de claves. Tales sistemas, denominados *sistemas de cripto de claves públicas*, serán posiblemente muy utilizados en un futuro próximo. Uno de los más conocidos

se basa en algunos de los algoritmos aritméticos que se han estudiado, por lo que se va a ver un poco más de cerca su modo de funcionamiento.

La idea de los sistemas de cripto de claves públicas es utilizar una «guía telefónica» de claves de cifra. La clave de cifra de cada uno (P) es de dominio público: la clave de una persona puede figurar, por ejemplo, en la guía, junto a su número de teléfono. Cada uno tiene también una clave secreta para descifrar: esta clave secreta (S) no la conoce nadie más. Para trasmitir un mensaje M , el emisor utiliza para cifrarlo la clave pública del receptor y luego lo transmite. El mensaje cifrado (texto cifrado) será $C = P(M)$. El receptor utiliza su clave privada para descifrar y leer el mensaje. Para que este sistema funcione, deben satisfacerse al menos las siguientes condiciones:

- (i) $S(P(M)) = M$ para todo mensaje M .
- (ii) Todos los pares (S, P) son diferentes.
- (iii) Obtener S a partir de P es tan difícil como leer M .
- (iv) Tanto S como P son fáciles de calcular.

La primera es una propiedad fundamental de la criptografía, la segunda y la tercera son para dar seguridad y la cuarta hace que los sistemas sean factibles de utilizar.

Este esquema general fue esbozado por W. Diffie y M. Hellman en 1976, pero sin proponer ningún método que cumpliera todas estas condiciones. Un método tal fue descubierto rápidamente por R. Rivest, A. Shamir y L. Adleman. Su esquema, que se conoce como *el sistema de cripto de claves públicas RSA*, está basado en la aplicación de algoritmos aritméticos sobre enteros muy grandes. La clave de cifrar P es el par de enteros (N, p) y la clave de descifrar S es el par de enteros (N, s) , donde s se mantiene secreto. Estos números deben ser muy grandes (típicamente, N puede tener 200 cifras y p y s 100). Los métodos de cifrar y descifrar son simples: primero se divide el mensaje en números menores que N (por ejemplo, tomando cada vez $\lg N$ bits de la cadena binaria correspondiente a la codificación por caracteres del mensaje). Luego se elevan estos números, de forma independiente, a una potencia módulo N : para *cifrar* un mensaje M (una parte del mensaje), se calcula $C = P(M) = M^P \bmod N$, y para *descifrar* un texto cifrado C , se calcula $M = S(C) = C^S \bmod N$. En el Capítulo 36 se estudiará cómo llevar a cabo este cálculo; aunque los cálculos con números de 200 cifras pueden ser engorrosos, el hecho de que sólo se necesite el resto de la división por N permite controlar el tamaño de los números, a pesar de que M^P y C^S sean prácticamente inconcebibles.

Propiedad 23.1 *En el sistema de cripto RSA, un mensaje se puede cifrar en un tiempo lineal.*

Para mensajes largos, la longitud de los números utilizados para las claves se puede considerar constante (un detalle de implementación). De forma similar, la exponentiación se efectúa en tiempo constante, puesto que no se permite que

los números sean mayores que esa longitud «constante». Es cierto que este argumento oculta muchas consideraciones de implementación relacionadas con las operaciones sobre grandes números; el coste de estas operaciones es, de hecho, un factor limitativo para la generalización de la aplicabilidad del método.■

Por tanto, se satisface la condición (iv) anterior y la condición (ii) es fácil de asegurar. Sólo queda estar seguros de que las variables de cifrar N , p y s se pueden escoger para que satisfagan las condiciones (i) y (iii). La demostración de esto necesita una presentación de la teoría de números que está fuera del alcance de este libro, pero es posible esbozar las ideas principales. En primer lugar hay que generar tres números «aleatorios» primos muy grandes (de 100 cifras aproximadamente): el mayor será s , denominándose a los otros dos x y y . Despues se escoge N igual al producto de x y y , y se elige p de modo que $ps \bmod (x-1)(y-1)=1$. Es posible demostrar que, eligiendo N , p y s de esta manera, se tiene que $M^{ps} \bmod N = M$ para todo mensaje M .

Por ejemplo, con la codificación estándar, el mensaje ATAQUE AL AMÉRICA corresponde al número de 36 cifras

012001172105000112000113011405030518

puesto que A es la primera letra (01) del alfabeto, T es la (20), etc. Para mantener el ejemplo dentro de unos límites razonables se consideran números primos de 2 cifras (y no de 100 como sería necesario): se toma $x = 47$, $y = 79$ y $s = 97$. Estos valores dan un $N = 3713$ (el producto de x y y) y $p = 37$ (el único entero que multiplicado por 97 da de resto 1 al dividirlo por 3588). Para cifrar el mensaje, se divide en paquetes de 4 cifras que se elevan a la potencia p (módulo N). Esto da la versión codificada

140403340803000108231215181505271657

Esto es, $0120^{37} \equiv 1404$, $0117^{37} \equiv 0334$, $2105^{37} \equiv 0803 \pmod{3713}$, etc. El proceso de descifrado es el mismo, pero utilizando s en lugar de p . Así, retrocediendo, se encuentra el mensaje original porque $1404^{97} \equiv 0120$, $0334^{97} \equiv 0117 \pmod{3713}$, etcétera.

La parte más importante de los cálculos es la codificación del mensaje, de acuerdo con la propiedad 23.1 anterior. Pero no hay sistema de cripto si no es posible calcular las variables clave. Aunque esto implica una teoría de números sofisticada y programas relativamente complejos para manipular números muy grandes, el tiempo de cálculo de las claves es normalmente inferior al cuadrado de su longitud (y no proporcional a su valor, lo que sería inaceptable).

Propiedad 23.2 *Las claves de un sistema de cripto RSA se pueden crear sin excesivos cálculos.*

Aquí se necesitan otra vez métodos que están fuera del alcance de este libro. Se

entiende que cada gran número primo se puede generar determinando primero un gran número aleatorio, y verificando después sucesivos números, comenzando a partir de aquél, hasta que se encuentre un primo. Un método simple permite efectuar un cálculo sobre un número aleatorio que, con probabilidad 1/2, «probará» que el número a comprobar no es primo. (Un número que no sea primo sobrevivirá a 20 aplicaciones de esta comprobación menos de una vez en un millón, y a 30 aplicaciones menos de una vez en mil millones.) El último paso consiste en calcular p : esto indica que una variante del algoritmo de Euclides (ver Capítulo 1) responde exactamente a las necesidades del problema.■

Recuérdese que la clave de descifrado s (y los factores x y y de N) se deben mantener en secreto, y que el éxito del método depende de que el criptoanalista no sea capaz de encontrar el valor de s , conociendo N y p . Para el ejemplo, es fácil encontrar que $3713 = 47 * 79$, pero si N es un número de 200 cifras, hay poca esperanza de encontrar sus factores. Esto es, parece difícil poder calcular s a partir del conocimiento de p (y N), aunque nadie ha sido capaz de *probar* que esto es así. Aparentemente encontrar p a partir de s necesita el conocimiento de x y de y , y parece inevitable descomponer a N en factores primos para calcular x y y . Pero esta descomposición de N es un problema muy difícil: el mejor algoritmo de descomposición conocido llevaría millones de años para descomponer un número de 200 cifras, utilizando la tecnología actual.

Una característica atractiva de los sistemas RSA es que los complicados cálculos que implican a N , p y s se llevan a cabo una sola vez por cada usuario que se suscribe al sistema, mientras que las operaciones cifrar y descifrar no implican más que dividir el mensaje y aplicar el simple procedimiento de exponentiación. Esta simplicidad de cálculo, combinada con las apropiadas características de los sistemas de cripto de claves públicas, hace a este sistema bastante conveniente para la comunicación del tipo confidencial, especialmente en redes y sistemas de computadoras.

El método RSA tiene sus inconvenientes: el procedimiento de exponentiación es bastante caro en los estándares de criptografía, y, lo que es peor, no es posible eliminar la eventualidad de que se puedan leer los mensajes cifrados utilizando este método. Esto es cierto en muchos sistemas de cripto: un método criptográfico debe resistir a un gran número de intentos de violación por parte de los criptoanalistas antes de que se pueda utilizar con total confianza.

Se han sugerido varios métodos para la implementación de sistemas de cripto de claves públicas. Los más interesantes están relacionados con una clase importante de problemas que generalmente se consideran como muy difíciles y que se estudiarán en el Capítulo 45. Estos sistemas de cripto poseen la interesante propiedad de que un ataque que tenga éxito puede proporcionar ideas sobre cómo resolver algunos de los difíciles e insolubles problemas (como el de la descomposición en factores primos en el método RSA). Esta relación entre la criptología y algunos dominios fundamentales de la investigación en la informática, junto con el potencial que significa la difusión de la criptografía de claves públicas, hacen de ella un campo muy activo de la investigación actual.

Ejercicios

1. Descifrar el siguiente mensaje, que se cifró con la cifra *Vigenere* utilizando como clave el patrón CAB (repetido tantas veces como sea necesario; alfabeto de 27 letras, con el espacio en blanco precediendo a la A): XOCQCQTHHWQUCCGCFJN
2. ¿Qué tabla se debe utilizar para *descifrar* mensajes que han sido cifrados utilizando el método de sustitución?
3. Suponiendo que se utiliza la cifra *Vigenere* con claves de dos caracteres para cifrar un mensaje relativamente largo, escribir un programa para adivinar la clave, partiendo de la hipótesis de que la frecuencia de aparición de los caracteres situados en posiciones impares debe ser aproximadamente igual a la de los caracteres de las posiciones pares.
4. Escribir procedimientos de cifrar y descifrar que utilicen la operación «o exclusivo» entre la versión binaria del mensaje y una secuencia binaria de uno de los generadores de números aleatorios de congruencia lineal del Capítulo 35.
5. Escribir un programa para romper el método del ejercicio anterior, suponiendo que se sabe que los primeros 10 caracteres del mensaje son espacios en blanco.
6. ¿Se podría cifrar un texto en claro mediante conjunciones «y» bit a bit entre mensaje y clave? Explicar por qué (o por qué no).
7. ¿Verdadero o falso? La criptografía de claves públicas facilita el envío del mismo mensaje a varios destinatarios. Explicar la respuesta.
8. ¿A qué es igual $P(S(M))$ en el método RSA de la criptografía de claves públicas?
9. El cifrado del tipo RSA puede implicar el cálculo de M^n , donde M puede ser un número de k cifras representado, por ejemplo, por un array de k enteros. ¿Cuántas operaciones se necesitan en este cálculo?
10. Implementar los procedimientos de cifrar/descifrar para el método RSA (suponiendo que s , p y N se representan por arrays de enteros de tamaño 25).

REFERENCIAS para el Procesamiento de cadenas

Las mejores fuentes para obtener más información sobre muchos de los temas que se han tratado en los capítulos de esta sección son las referencias originales. El artículo de Knuth, Morris y Pratt de 1977, los de Boyer y Moore de 1977 y de Karp y Rabin de 1981 constituyen la base de la mayor parte del material del Capítulo 19. El trabajo de Thompson de 1968 es la base del método de reconocimiento de patrones descritos por expresiones regulares de los Capítulos 20 y 21. El artículo de Huffman de 1952 es anterior a muchas de las consideraciones algorítmicas hechas aquí, pero todavía es una lectura de interés. Rivest, Shamir y Adleman describen completamente la implementación y la aplicación de su sistema de cripto de claves públicas en su trabajo de 1978.

El libro de Standish es una buena referencia para muchos de los temas cubiertos por esta sección, especialmente en los Capítulos 19, 22 y 23. Ese libro trata también algunas representaciones y algoritmos prácticos no descritos aquí. El análisis sintáctico y la compilación son para muchos el corazón de la informática: se ha investigado su relación con los algoritmos, pero su relación con los lenguajes de programación, la teoría de la información y otras áreas es mucho más importante. Gran parte de los aspectos algorítmicos se ha estudiado con gran detalle. La referencia estándar sobre este tema es el libro de Aho, Sethi y Ullman.

Como es obvio, la literatura pública sobre criptografía es bastante escasa. Sin embargo, se puede encontrar mucha información general sobre el tema en los libros de Kahn y Konheim.

- A. V. Aho, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, Tools*, Addison-Wesley, Reading, MA, 1986. (Existe versión en español por Addison-Wesley Iberoamericana *N. del E.*)
- R. S. Boyer y J. S. Moore, «A fast string searching algorithm», *Communications of the ACM*, **20**, 10 (octubre, 1977).
- D. A. Huffman, «A method for the construction of minimum-redundancy codes», *Proceedings of the IRE*, **40** (1952).
- D. Kahn, *The Codebreakers*, Macmillan, New York, 1967.
- R. M. Karp y M. O. Rabin, «Efficient Randomized Pattern-Matching Algorithms», Technical Report TR-31-81, Aiken Comput. Lab., Harvard U., Cambridge, MA 1981.
- D. E. Knuth, J. H. Morris y V. R. Pratt, «Fast pattern matching in strings», *SIAM Journal on Computing*, **6**, 2 (junio, 1977).
- A. G. Konheim, *Cryptography: A Primer*, John Wiley & Sons, New York, 1981.
- R. L. Rivest, A. Shamir y L. Alderman, «A method for obtaining digital signatures and public-key cryptosystems», *Communications of the ACM*, **21**, 2 (febrero, 1978).
- T. A. Standish, *Data Structure Techniques*, Addison-Wesley, Reading, MA, 1980.
- K. Thompson, «Regular expression search algorithm», *Communications of the ACM*, **11**, 6 (junio, 1968).

Algoritmos geométricos

Métodos geométricos elementales

Las computadoras se están utilizando cada día más para resolver problemas a gran escala que son inherentemente geométricos. Los objetos geométricos, tales como puntos, líneas y polígonos constituyen la base de una gran variedad de aplicaciones importantes y conducen a un interesante conjunto de problemas y algoritmos.

Los algoritmos geométricos son importantes en sistemas de diseño y análisis de modelos de objetos físicos, que pueden ser desde edificios y automóviles hasta circuitos integrados a escala muy grande. Un diseñador que trabaja con un objeto físico posee una intuición geométrica que resulta difícil de aplicar en una representación por computadora. Otras muchas aplicaciones procesan datos geométricos de forma directa. Por ejemplo, un esquema político de «manipulación del censo electoral», que sirva para dividir un distrito en áreas de igual población (y que satisfaga otros criterios, como colocar a todos los miembros del otro partido en una misma zona), es un sofisticado algoritmo geométrico. Otras aplicaciones son de tipo matemático o estadístico, campos en los que muchos tipos de problemas pueden ser naturalmente puestos en una representación geométrica.

La mayoría de los algoritmos que se han estudiado utilizan texto y números, que se representan y se procesan de forma natural en la mayoría de los entornos de programación. De hecho, las operaciones primitivas necesarias se implantan en el hardware de la mayoría de los sistemas de computadoras. Se verá que la situación es diferente en el caso de los problemas geométricos: incluso las operaciones más elementales con puntos y líneas pueden ser un reto en términos informáticos.

Los problemas geométricos son muy fáciles de visualizar, pero eso puede ser un inconveniente. Muchos problemas, que una persona puede resolver instantáneamente mirando un papel (por ejemplo: ¿está un punto dentro de un polí-

gono?), requieren programas de computadora que no son triviales. En el caso de problemas más complicados, como en muchas otras aplicaciones, el método de resolución apropiado para su implantación en una computadora puede ser bastante diferente del método de resolución adecuado para una persona.

Se podría pensar que los algoritmos geométricos deben tener una larga historia, debido a la naturaleza constructiva de la antigua geometría y porque las aplicaciones útiles están muy difundidas, pero, en realidad, la mayor parte de los avances en este campo han sido bastante recientes. Sin embargo, el trabajo de los antiguos matemáticos resulta a menudo útil para el desarrollo de algoritmos para las modernas computadoras. El campo de los algoritmos geométricos es interesante de estudiar debido a su fuerte contexto histórico, porque aún se están desarrollando nuevos algoritmos fundamentales y porque numerosas aplicaciones importantes a gran escala necesitan estos algoritmos.

Puntos, líneas y polígonos

La mayoría de los programas que se estudiarán operan sobre objetos geométricos simples definidos en un espacio bidimensional, si bien se tendrá en cuenta algunos algoritmos para más dimensiones. El objeto fundamental es el *punto*, al que se considera como un par de enteros —las «coordenadas» del punto en el sistema cartesiano habitual—. Una *línea* es un par de puntos, que se supone que están unidos por un segmento de línea recta. Un *polígono* es una lista de puntos: se supone que puntos sucesivos están unidos por líneas y que el primer punto está conectado al último, para formar una figura cerrada.

Para poder trabajar con estos objetos geométricos se necesita decidir cómo representarlos. Normalmente se utiliza un array para los polígonos, aunque también se puede usar una lista enlazada o alguna otra representación cuando sea apropiado. La mayoría de los programas utilizarán las siguientes representaciones:

```
struct punto {int x, y; char c; };
struct linea { struct punto p1, p2; };
struct punto poligono[Nmax];
```

Hay que destacar que los puntos sólo pueden tener coordenadas enteras. También se podría utilizar una representación en coma flotante. El uso de coordenadas enteras hace que los algoritmos sean algo más sencillos y más eficaces, y no es una restricción tan estricta como podría parecer. Como ya se mencionó en el Capítulo 2, la utilización de enteros siempre que sea posible puede ahorrar bastante tiempo en muchos entornos de computación, ya que los cálculos con enteros son mucho más eficientes que las operaciones en coma flotante. Por tanto, cuando se pueda conseguir un propósito utilizando solamente enteros, sin introducir demasiadas complicaciones extra, éste será el camino a elegir.

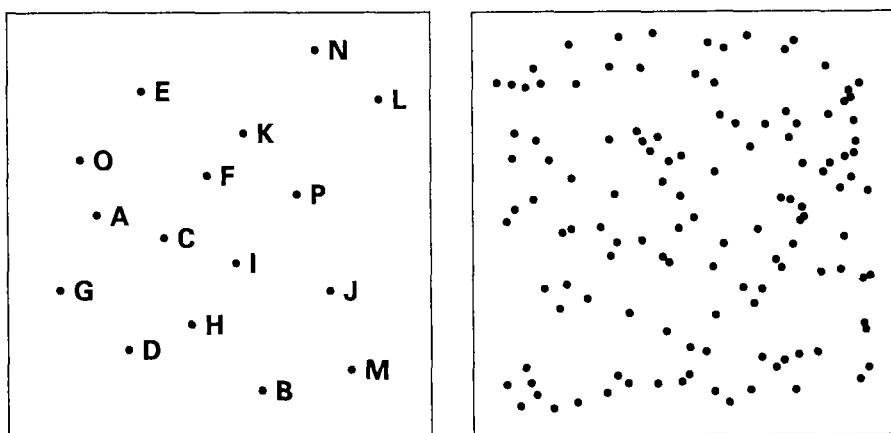


Figura 24.1 Conjuntos de puntos para algoritmos geométricos.

Se representarán los objetos geométricos más complicados en función de estos componentes básicos. Por ejemplo, los polígonos se representarán como arrays de puntos. Se puede advertir que el uso de arrays de líneas supondría que cada punto del polígono estaría incluido dos veces (aunque ésta podría ser la representación natural para determinados algoritmos). Además, en algunas aplicaciones resulta útil incluir información adicional asociada a cada punto o línea; se puede hacer esto añadiendo un campo *info* en los registros.

Se utilizará el conjunto de puntos mostrado en la Figura 24.1 para ilustrar las operaciones de varios algoritmos geométricos. Los 16 puntos de la izquierda están etiquetados con letras que servirán de referencia en las explicaciones de los ejemplos, y poseen las coordenadas enteras que aparecen en la Figura 24.2. (Las letras de las etiquetas se han asignado en el orden en el que se supone se introducen los puntos.) Por lo regular, los programas no tienen motivos para hacer referencia a los puntos por su «nombre»; éstos simplemente se almacenan en un array y se refieren usando un índice. El orden en el que se almacenan los puntos dentro del array puede ser importante en algunos programas: de hecho, el objetivo de algunos algoritmos geométricos consiste en «ordenar» los puntos de una forma determinada. En la parte derecha de la Figura 24.1 hay

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
x	3	11	6	4	5	8	1	7	9	14	10	16	15	13	12
y	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12

Figura 24.2 Coordenadas de los puntos del pequeño conjunto de ejemplo (ver Figura 24.1).

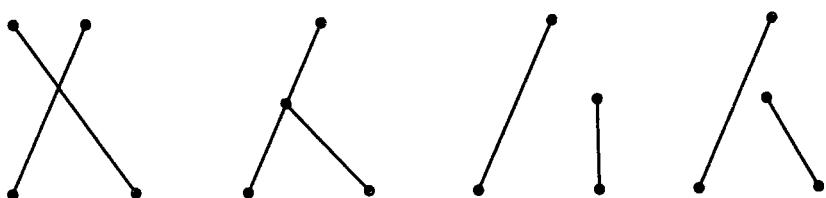


Figura 24.3 Comprobación de la intersección de segmentos: cuatro casos.

128 puntos generados aleatoriamente, con coordenadas enteras que varían entre 0 y 1000.

Un programa típico gestiona un array de p puntos y simplemente lee N pares de enteros, asignando el primer par a las coordenadas x e y de $p[1]$, el segundo par a $p[2]$, etc. Cuando p representa a un polígono, a veces es conveniente mantener valores «centinelas» $p[0]=p[N]$ y $p[N+1]=p[1]$.

Intersección de segmentos de líneas

Como primer problema geométrico elemental, se considerará si dos segmentos determinados se cortan o no. La Figura 24.3 ilustra algunas de las situaciones que se pueden dar. En el primer caso, los segmentos se cortan. En el segundo, el extremo de un segmento está situado en el otro segmento. Se considerará que esto es una intersección, suponiendo que los segmentos son «cerrados» (los extremos forman parte de los segmentos); por tanto, los segmentos que poseen un extremo en común se cortan. En los dos últimos casos de la Figura 24.3 los segmentos no se cortan, pero los casos difieren si se considera el punto de intersección de las líneas definidas por los segmentos. En el cuarto caso, este punto de intersección se encuentra en uno de los segmentos; en el tercero caso no es así. O también, las líneas podrían ser paralelas (un caso especial, que aparece con frecuencia, ocurre cuando uno de los segmentos, o ambos, son puntos).

La forma más directa de solucionar este problema consiste en encontrar el punto de intersección de las líneas definidas por los segmentos y comprobar después si este punto de intersección está situado entre los extremos de ambos segmentos. Otro método sencillo se basa en una herramienta que será de utilidad más adelante, por lo que se estudiará con más detalle. Dados tres puntos, se quiere saber si, al ir del primero al segundo y de éste al tercero, el movimiento es en el sentido contrario al de las manecillas del reloj. Por ejemplo: para los puntos A, B y C de la Figura 24.1, la respuesta es afirmativa, pero para los puntos A, B y D, la respuesta es negativa. Esta función es sencilla de calcular a partir de las ecuaciones de las líneas, como se muestra a continuación:

```
int ccw(struct punto p0,
```

```

    struct punto p1,
    struct punto p2 )
{
    int dx1, dx2, dy1, dy2;
    dx1 = p1.x - p0.x; dy1 = p1.y - p0.y;
    dx2 = p2.x - p0.x; dy2 = p2.y - p0.y;
    if (dx1*dy2 > dy1*dx2) return +1;
    if (dx1*dy2 > dy1*dx2) return -1;
    if ((dx1*dx2 < 0) (dy1*dy2 < 0)) return -1;
    if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2))
                                return +1;
    return 0;
}

```

Para entender cómo funciona el programa, se supone en primer lugar que las cantidades dx_1 , dx_2 , dy_1 y dy_2 son positivas. A continuación se observa que la pendiente de la línea que une p_0 y p_1 es dy_1/dx_1 y la pendiente de la línea que une p_0 y p_2 es dy_2/dx_2 . Ahora, si la pendiente de la segunda línea es mayor que la pendiente de la primera, se necesita un desplazamiento hacia la «izquierda» (en sentido contrario al de las manecillas del reloj) para ir de p_0 a p_1 y a p_2 ; si es menor, se necesita un desplazamiento a la «derecha» (en el sentido de las manecillas del reloj). La comparación de pendientes en el programa es algo inconveniente, puesto que las líneas podrían ser verticales (dx_1 o dx_2 podrían ser 0): para evitarlo, se multiplica dx_1*dx_2 . Se puede ver que no es necesario que las pendientes sean positivas para que esta prueba funcione correctamente —la demostración se deja como un instructivo ejercicio—.

Pero existe una omisión crucial en la descripción anterior: se ignoran los casos en los que las pendientes son iguales (los tres puntos son colineales). En estos casos, se puede pensar en varias formas de definir la función ccw . La opción que se ha elegido consiste en hacer que la función tenga tres valores: en vez de utilizar la notación estándar, en la que el valor devuelto es cero o un valor distinto de cero, se utilizan los valores 1 y -1, reservando el valor 0 para el caso en el que p_2 está sobre el segmento que une p_0 y p_1 . Si los puntos son colineales, y p_0 está situado entre p_2 y p_1 , se hace que ccw devuelva -1; si p_2 está situado entre p_0 y p_1 , se hace que ccw devuelva 0; y si p_1 está situado entre p_0 y p_2 , se hace que ccw devuelva 1. Se verá que este convenio simplifica la codificación de las funciones que utilizan ccw , en este capítulo y en el siguiente.

Con estas definiciones se puede implantar rápidamente la función *intersec*. Si los dos extremos de cada línea están en diferentes «dados» (tienen distintos valores de ccw) respecto a la otra, entonces las líneas deben cortarse:

```

int intersec(struct linea l1, struct linea l2)
{

```

```

    return ((ccw(l1.p1, l1.p2, l2.p1)
            *ccw(l1.p1, l1.p2, l2.p2)) <= 0)
    && ((ccw(l2.p1, l2.p2, l1.p1)
            *ccw(l2.p1, l2.p2, l1.p1)) <= 0);
}

```

Esta solución parece que supone la realización de una gran cantidad de cálculos para un problema tan sencillo. Se anima al lector a que intente encontrar una solución más simple, asegurándose de que funciona en todos los casos. Por ejemplo, si los cuatro puntos son colineales, existen seis casos distintos (sin contar las situaciones en las que hay puntos coincidentes), de los cuales sólo cuatro son intersecciones. Los casos especiales como éstos son el problema de los algoritmos geométricos: no se pueden evitar, pero se puede minimizar su impacto utilizando primitivas como ccw.

Si hay implicadas muchas líneas, la situación pasa a ser mucho más complicada. En el Capítulo 27 se verá un sofisticado algoritmo que determina si se cortan dos líneas cualesquiera de un conjunto de N líneas.

Camino cerrado simple

Para poder saborear los problemas que se refieren a conjuntos de puntos, considérese el problema de encontrar, a partir de un conjunto de N puntos, un camino que no se corte a sí mismo, que recorra todos los puntos y que vuelva al punto inicial. Tal camino se denomina *camino cerrado simple*. Es posible imaginar muchas aplicaciones para esto: los puntos podrían representar casas, y el camino puede ser la ruta que seguiría un cartero para visitar todas las casas sin cruzar su propio trayecto. O, simplemente, se podría buscar una forma razonable de dibujar los puntos usando un plotter mecánico. Este problema es elemental, porque sólo busca cualquier camino cerrado que conecte los puntos. El problema de buscar el mejor de los caminos, conocido como el *problema del vendedor ambulante*, es mucho, muchísimo más difícil, y se abordará con cierto detalle en los últimos capítulos de este libro. En el siguiente capítulo se considerará un problema relacionado con él, pero mucho más sencillo: encontrar el camino más corto que envuelve a un determinado conjunto de N puntos. En el Capítulo 31 se verá cómo encontrar la mejor forma de «conectar» un conjunto de puntos.

Una forma sencilla de resolver este problema elemental es la siguiente: se selecciona uno de los puntos, que servirá como «pivot». Despues se calcula el ángulo de las líneas que unen el pivot con cada uno de los puntos del conjunto según la dirección horizontal positiva (esto es parte de las coordenadas polares de cada punto del conjunto, con el pivot como origen). A continuación se ordenan los puntos según el ángulo calculado. Por último se conectan los puntos

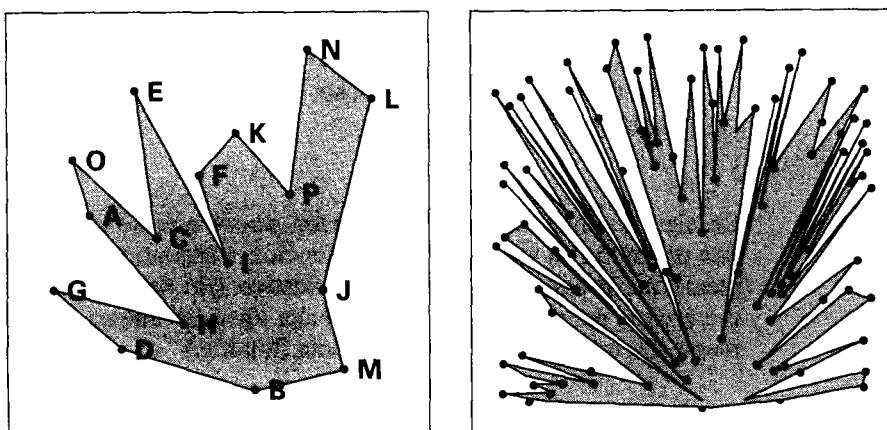


Figura 24.4 Caminos cerrados simples.

adyacentes. El resultado es un camino cerrado simple que conecta todos los puntos, como se muestra en la Figura 24.4 para los puntos de la Figura 24.1. En el pequeño conjunto de puntos, se utiliza B como pivote: si los puntos se recorren en el orden

B M J L N P K F I E C O A H G D B

se dibujará un polígono cerrado simple.

Si dx y dy son las distancias en los ejes x e y , desde el punto pivote a cualquier otro punto, el ángulo buscado en este algoritmo es $\tan^{-1} dy/dx$. Aunque la arcotangente es una función incorporada en C++ (y en algunos otros entornos de programación), es probable que sea lenta y que calcule además al menos dos condiciones molestas para el cálculo: si dx es cero y en qué cuadrante está el punto. Puesto que en este algoritmo el ángulo sólo se utiliza para la ordenación, tiene sentido utilizar una función que sea mucho más sencilla de calcular, pero que tenga las mismas propiedades de ordenación que la arcotangente (de forma que, al ordenar, se obtengan los mismos resultados). Una buena candidata para esta función es simplemente $dy/(dy+dx)$. Aún sigue siendo necesario comprobar las condiciones excepcionales, pero resulta más sencillo. El siguiente programa devuelve un número entre 0 y 360 que no es el ángulo formado por $p1$ y $p2$ con la horizontal, pero que tiene las mismas propiedades de ordenación:

```
float theta( struct punto p1, struct punto p2)
{
    int dx, dy, ax, ay;
    float t;
```

```

dx = p2.x - p1.x; ax = abs(dx);
dy = p2.y - p1.y; ay = abs(dy);
t = (ax+ay == 0) ? 0 : (float) dy/(ax+ay);
if (dx < 0) t = 2-t; else if (dy < 0) t = 4+t;
return t*90.0;
}

```

En algunos entornos de programación puede que no merezca la pena utilizar este programa en sustitución de las funciones trigonométricas estándar; en otros, puede que se consiga un ahorro significativo. (En determinados casos, puede que merezca la pena hacer que theta tenga un valor entero, para evitar completamente el empleo de números en coma flotante.)

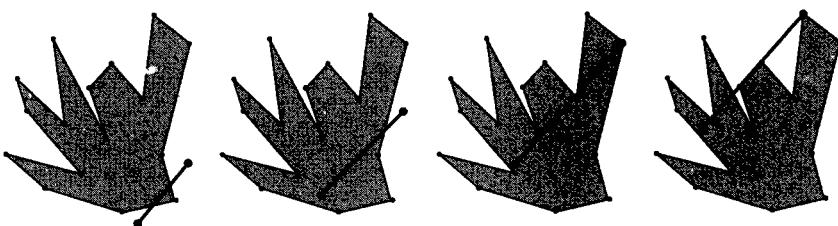


Figura 24.5 Casos a tener en cuenta en el algoritmo del punto en el polígono.

Inclusión en un polígono

El siguiente problema que se va a considerar es muy natural: dados un punto y un polígono representado como un array de puntos, determinar si el punto está dentro o fuera del polígono. De inmediato se puede ver una solución directa a este problema: se dibuja una línea larga desde el punto, en cualquier dirección (lo suficientemente larga como para garantizar que el otro extremo esté situado fuera del polígono), y se cuenta el número de líneas del polígono a las que corta. Si el número es impar, el punto debe estar en el interior; si es par, el punto es exterior. Esto se comprueba fácilmente viendo lo que sucede al acercarse desde el extremo exterior: tras la primera intersección, se está dentro; tras la segunda, de nuevo se está fuera, etc. Si se hace esto un número par de veces, el punto al que se llega (el punto original) debe estar en el exterior.

No obstante, la situación no es tan simple, ya que se pueden producir algunas intersecciones justo en los vértices del polígono dado. La Figura 24.5 muestra algunas de las situaciones que se deben tener en cuenta. La primera es un caso directo de un punto exterior al polígono; la segunda es un caso directo de punto interior; en el tercer caso, la línea de prueba sale del polígono por un vértice (tras tocar otros dos vértices); y en el cuarto caso, la línea de prueba coin-

cide con uno de los lados del polígono antes de salir. En alguno de los casos en los que la línea de prueba corta a un vértice, debería contar como una intersección con el polígono; en otros casos, no debería contar (o debería contar como dos intersecciones). El lector se puede entretenar intentando encontrar una sencilla prueba para distinguir estos casos antes de continuar leyendo.

La necesidad de tener en cuenta los casos en los que las líneas de prueba pasan por los vértices obliga a hacer algo más que contar los lados del polígono que se cortan con la línea de prueba. En esencia, se desea desplazarse alrededor del polígono, incrementando el contador de intersecciones siempre que se pase de un lado de la línea de prueba a otro. Una forma de implantar esto consiste simplemente en ignorar los puntos por los que pasa la línea de prueba, como en el siguiente programa:

```
int interior(struct punto t, struct punto[], int N)
{
    int i, cont = 0, j = 0;
    struct linea lt, lp;
    p[0] = p[N]; p[N+1] = p[1];
    lt.p1 = t; lt.p2 = t; lt.p2.x = INT_MAX;
    for (i=1; i<=N; i++)
    {
        lp.p1= p[i]; lp.p2= p[i];
        if (!intersec(lp,lt))
        {
            lp.p2= p[j]; j=i;
            if (intersec(lp,lt)) cont++;
        }
    }
    return cont & 1;
}
```

Este programa hace uso de una línea horizontal de prueba para simplificar los cálculos (pueden imaginarse los diagramas de la Figura 24.5 rotados 45 grados). La variable *j* se utiliza como índice del último punto del polígono que se sabe que no va a estar sobre la línea de prueba. El programa supone que *p[1]* es el punto que tiene la menor coordenada *x* de entre todos los puntos con la menor coordenada *y*, de modo que si *p[1]* está en la línea de prueba, *p[0]* no lo puede estar también. El mismo polígono se puede representar mediante *N* diferentes arrays *p*, pero como se puede comprobar, a veces resulta conveniente fijar una regla estándar para *p[1]*. (Por ejemplo, esta misma regla es útil para usar *p[1]* como pivote en el procedimiento sugerido anteriormente para el cálculo del camino cerrado simple.) Si el siguiente punto del polígono que no está en la línea de prueba está en el mismo lado de la línea de prueba que el punto *j*-ésimo, no

se necesita incrementar el contador de intersecciones (cont); en caso contrario, se tiene una intersección. Si se desea, el lector puede comprobar que este algoritmo funciona adecuadamente en los casos de la Figura 24.5.

Si el polígono sólo tiene tres o cuatro caras, como sucede en muchas aplicaciones, no es apropiado usar un programa tan complejo: un procedimiento más simple basado en llamadas a ccw será más adecuado. Otro caso especial importante es el *polígono convexo*, que se estudiará en el siguiente capítulo, y que tiene la propiedad de que ninguna línea de prueba puede tener más de dos intersecciones con el polígono. En este caso se puede utilizar un procedimiento como el de la búsqueda binaria para determinar en $O(\log N)$ pasos si el punto está o no dentro del polígono.

Perspectiva

De los pocos ejemplos anteriores, debería estar claro que resulta fácil subestimar la dificultad de la resolución de un determinado problema geométrico utilizando una computadora. Existen otros muchos cálculos geométricos elementales que no se han estudiado. Por ejemplo, un ejercicio interesante podría ser la realización de un programa que calcule el área de un polígono. Los problemas vistos hasta el momento constituyen unas herramientas básicas que serán útiles para solucionar algunos problemas más complicados. No obstante, la variedad de problemas y algoritmos a tener en cuenta es tan grande que se debe restringir el estudio a algunos ejemplos seleccionados que resuelvan problemas fundamentales y que estén relacionados con los algoritmos que se han visto.

Algunos de los algoritmos que se estudiarán implican la construcción de estructuras geométricas a partir de un determinado conjunto de puntos. El «polígono cerrado simple» es un ejemplo elemental de esto. Se necesitará decidir las representaciones apropiadas para tales estructuras, desarrollar los algoritmos para construirlas y estudiar su uso en aplicaciones concretas. Como siempre, estas consideraciones están interrelacionadas. Por ejemplo, el algoritmo usado en el procedimiento *interior* de este capítulo depende totalmente de la representación del polígono cerrado simple como conjunto ordenado de puntos (en lugar de, por ejemplo, un conjunto desordenado de líneas).

Como es habitual, la característica de abstracción de datos de C++ proporciona una forma conveniente de ofrecer las diversas opciones de representación de las aplicaciones. Por otra parte, las aplicaciones geométricas pueden beneficiarse de la estructura jerárquica de clases que ofrece C++ para organizar de forma apropiada todos los tipos de objetos y las operaciones sobre los mismos que se deben implementar. De hecho, los textos sobre C++ a menudo utilizan objetos geométricos para ilustrar las ventajas de este enfoque. En este libro no se verán estos temas con mucho mayor detenimiento, ya que, desde un punto de vista *algorítmico*, las operaciones necesarias suelen ser demasiado elementales (ejemplo: dibujar o rotar una figura), o bien demasiado complicadas (ejem-

plo: calcular la intersección de dos polígonos). La realización de un paquete de software apropiado que soporte búsquedas, intersecciones y otras operaciones aplicables a un conjunto dinámico de puntos, líneas y polígonos excede los propósitos de este libro, si bien se intentará considerar de qué forma se debería implantar eficazmente las operaciones más importantes.

Muchos de los algoritmos que se estudian utilizan una *búsqueda geométrica*: se desea conocer qué puntos de un determinado conjunto están cerca de un punto dado, o qué puntos están dentro de un rectángulo dado, o cuáles son los puntos que están situados más cerca entre sí. La mayoría de los algoritmos apropiados para tales problemas de búsqueda están íntimamente relacionados con los algoritmos de búsqueda estudiados en los Capítulos 14 a 17. El paralelismo será bastante evidente.

Se han analizado pocos algoritmos geométricos para que se puedan formular valoraciones precisas sobre sus características de rendimiento relativo. Como se ha visto hasta ahora, el tiempo de ejecución de un algoritmo geométrico puede depender de muchos factores. La distribución de los propios puntos, el orden en que se introducen y la utilización de funciones trigonométricas pueden, en su conjunto, afectar de forma significativa al tiempo de ejecución de los algoritmos geométricos. No obstante, como viene siendo habitual en tales situaciones, se poseen datos empíricos que indican cuáles son los buenos algoritmos para determinadas aplicaciones. Además, muchos de los algoritmos se derivan de estudios complejos y han sido diseñados para tener buenos rendimientos en el peor caso.

Ejercicios

1. Indicar el valor de ccw para los tres casos en los que dos de los puntos son idénticos (y el tercero es distinto), y para el caso en el que los tres puntos son idénticos.
2. Encontrar un algoritmo rápido que determine si dos segmentos son paralelos, sin utilizar ninguna división.
3. Encontrar un algoritmo rápido que determine si cuatro segmentos forman un cuadrado, sin utilizar ninguna división.
4. Dado un array de líneas, ¿cómo se comprobaría si forman un polígono cerrado simple?
5. Dibujar los polígonos cerrados simples que resultan de utilizar los puntos A, C y D de la Figura 24.1 como «pivotes» según el método descrito en el texto.
6. Suponiendo que se utiliza un punto arbitrario como «pivot» en el método descrito en el texto para calcular un polígono cerrado simple, indicar las condiciones que debe satisfacer dicho punto para que el método funcione correctamente.

7. ¿Qué valor devuelve la función `intersec` cuando se la llama utilizando dos copias del mismo segmento?
8. ¿Considera la función `interior` que un vértice del polígono es interior, o, por el contrario, lo toma como exterior?
9. ¿Cuál es el máximo valor que puede tomar la variable `cont` cuando se ejecuta `interior` con un polígono de N vértices? Mostrar un ejemplo que apoye la respuesta.
10. Escribir un programa eficaz que determine si un punto dado está en el interior de un determinado cuadrilátero.

Obtención del cerco convexo

A veces, cuando hay que procesar un elevado número de puntos, lo que interesa es conocer los límites del conjunto de dichos puntos. Al observar en un diagrama un conjunto de puntos dibujados en el plano, normalmente no hay problema en distinguir los puntos que están «dentro» del conjunto de los que se encuentran en los bordes. Esta distinción es una característica fundamental de los conjuntos de puntos; en este capítulo se verá cómo se pueden caracterizar de forma precisa, examinando algoritmos que distinguen los puntos que conforman el «límite natural».

El método matemático utilizado para la descripción del límite natural de un conjunto de puntos depende de una propiedad geométrica denominada *convexidad*. Se trata de un concepto sencillo que posiblemente ya conozca el lector: un *polígono convexo* posee la propiedad de que cualquier línea que una dos puntos cualesquiera del interior del polígono estará dentro del mismo. Por ejemplo, el «polígono cerrado simple» que se calculó en el capítulo anterior es, decididamente, no convexo; por su parte, todos los triángulos y rectángulos son convexos.

El nombre matemático del límite natural de un conjunto de puntos es *cerco convexo*. Se define el cerco convexo de un conjunto de puntos del plano como el polígono convexo más pequeño que los contiene a todos. El cerco convexo es el camino más pequeño que envuelve los puntos. Una propiedad obvia y fácil de probar del cerco convexo es que los vértices del polígono convexo que definen el cerco son puntos pertenecientes al conjunto original de puntos. Dados N puntos, algunos de ellos forman un polígono convexo, dentro del cual están contenidos todos los demás. El problema consiste en encontrar esos puntos. Se han desarrollado numerosos algoritmos para encontrar el cerco convexo: en este capítulo se examinarán algunos de los más importantes.

La Figura 25.1 muestra los conjuntos de puntos de ejemplo de la Figura 24.1

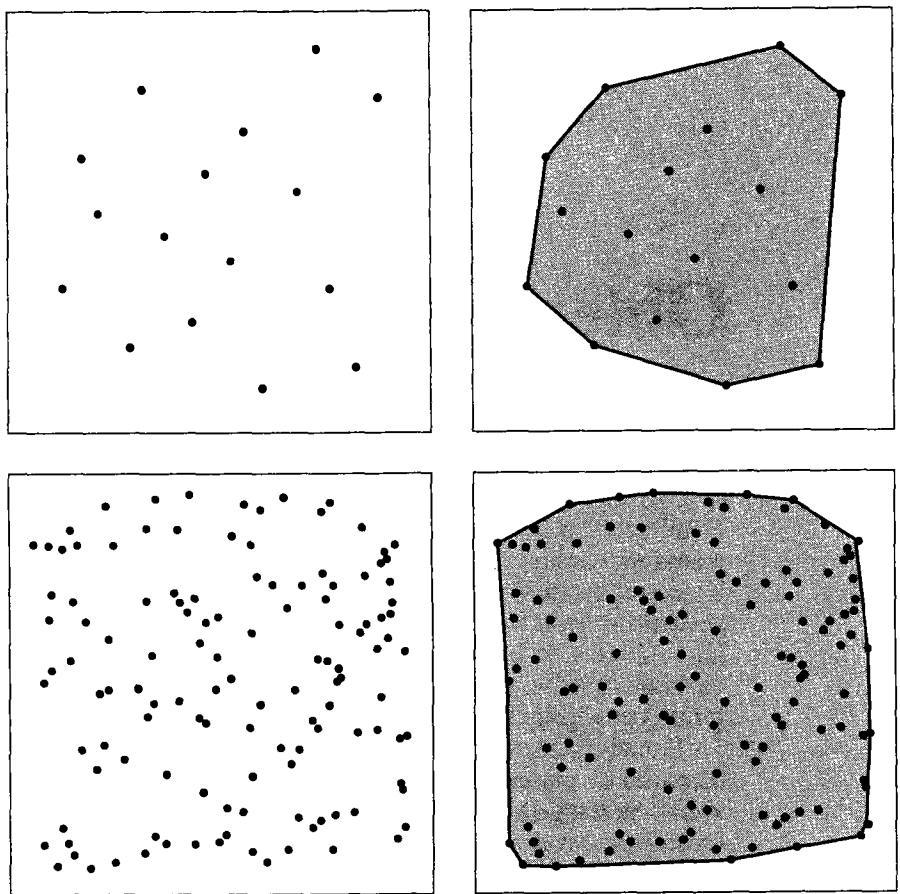


Figura 25.1 Cercos convексos de los puntos de la Figura 24.1.

y sus cercos convexos. Hay 8 puntos en el cerco del conjunto pequeño y 15 en el cerco del conjunto grande. En general, el cerco convexo puede contener, como mínimo, desde tres puntos (si los tres puntos forman un triángulo que contiene a los demás), hasta, como máximo, todos los puntos (si todos ellos están situados en el cerco convexo, en cuyo caso los puntos constituyen su propio cerco convexo). El número de puntos del cerco convexo de un conjunto de puntos «aleatorio» está comprendido entre esos extremos, como se verá a continuación. Algunos algoritmos funcionan bien cuando hay muchos puntos en el cerco convexo; otros funcionan mejor cuando sólo hay unos pocos.

Una propiedad fundamental del cerco convexo es que cualquier línea exterior al cerco, al desplazarla hacia él en cualquier dirección, tocará al menos uno de los puntos vértice. (Ésta es una forma alternativa de definir el cerco: es el

subconjunto del conjunto de puntos que puede ser alcanzado por alguna línea que se mueva con algún ángulo desde el infinito.) En particular, es fácil encontrar algunos pocos puntos con garantías de que estén en el cerco aplicando esta regla con líneas horizontales y verticales: los puntos que tengan las coordenadas x e y más pequeñas y más grandes pertenecen al cerco. Este hecho se utiliza como punto de partida de los algoritmos a estudiar.

Reglas del juego

La entrada de un algoritmo de búsqueda del cerco convexo es, por supuesto, un array de puntos; se puede utilizar el tipo punto definido en el capítulo anterior. La salida es un polígono, también representado como un array de puntos, que tiene la particularidad de que, al ir uniendo los puntos en el orden en que aparecen en el array, se dibuja el polígono. Pensándolo bien, esto puede parecer que requiere una condición adicional de ordenación en el cálculo del cerco convexo (¿por qué no devolver los puntos del cerco en cualquier orden?), pero, obviamente, la salida en forma ordenada resulta más útil, y ya se ha visto que los cálculos sin orden no son más fáciles de realizar. En todos los algoritmos que se estudiarán es conveniente realizar los cálculos *in situ*: el array utilizado para el conjunto de puntos original también se utiliza para guardar el resultado. Los algoritmos simplemente reordenan los puntos del array original de modo que el cerco convexo aparezca, ordenado, en las M primeras posiciones.

A la vista de la descripción anterior, queda claro que el cálculo del cerco convexo está íntimamente relacionado con la ordenación. De hecho, es posible utilizar un algoritmo de cerco convexo para realizar una ordenación de la siguiente forma. Dados N números a ordenar, se convierten en puntos (en coordenadas polares), considerando los números como ángulos (adecuadamente normalizados) con un radio fijo para todos los puntos. El cerco convexo de este conjunto de puntos es un polígono de N lados que contiene todos los puntos. Puesto que la salida debe estar ordenada según el orden de aparición de los puntos en el polígono, se puede utilizar para hallar el orden adecuado de los valores originales (recordando que los datos introducidos estaban desordenados). Esto no constituye una prueba formal de que el cálculo de un cerco convexo no es más sencillo de realizar que una ordenación, porque, por ejemplo, se debe tener en cuenta el coste que ocasiona el uso de las funciones trigonométricas necesarias para la conversión de los números en puntos del polígono. El comparar algoritmos de cerco convexo (que implican la utilización de operaciones trigonométricas) con algoritmos de ordenación (que implican comparaciones entre claves) es casi como comparar manzanas con naranjas, pese a lo cual se ha visto que cualquier algoritmo de cerco convexo requiere unas $N \log N$ operaciones, lo mismo que las ordenaciones (incluso siendo probable que las operaciones permitidas sean muy diferentes). Resulta útil considerar el cálculo de un cerco con-

vexo como una especie de «ordenación bidimensional», ya que en el estudio de algoritmos de cálculo de cercos convexos surgen frecuentes paralelismos con los algoritmos de ordenación.

De hecho, los algoritmos que se estudiarán muestran que hallar un cerco convexo no es más arduo que realizar una ordenación: existen varios algoritmos que, en el peor caso, se ejecutan en un tiempo proporcional a $N \log N$. Muchos de los algoritmos tienden a utilizar incluso menos tiempo en conjuntos de puntos reales, debido a que su tiempo de ejecución depende de la distribución de tales puntos, así como del número de puntos que forman el cerco.

Como con todos los algoritmos geométricos, hay que prestar alguna atención a los casos degenerados que probablemente aparezcan en la entrada. Por ejemplo, ¿cuál es el cerco convexo de un conjunto de puntos que están alineados? Dependiendo de la aplicación, podrían ser todos los puntos, o sólo los dos extremos, o quizás también valdría cualquier conjunto que incluya los dos puntos extremos. Aunque éste puede parecer un ejemplo extremo, no sería inusual que más de dos puntos se encuentren situados en uno de los segmentos que definen el cerco de un conjunto de puntos. En los siguientes algoritmos no se insistirá en incluir los puntos que estén situados en uno de los lados del cerco, ya que, en general, esto supone más trabajo (si bien, cuando proceda, se indicará cómo se podría hacer). Por otro lado, tampoco se insistirá en que se deben omitir estos puntos, ya que, si se desea, esta condición se podría comprobar con posterioridad.

Envolventes

El algoritmo más natural de cerco convexo, que se asemeja al método que utilizaría una persona para dibujar el cerco convexo de un conjunto de puntos, es una forma sistemática de «envolver» el conjunto de puntos. Empezando por algún punto que pertenezca con seguridad al cerco convexo (por ejemplo, el que tenga la coordenada y más pequeña), se traza una línea recta y se gira, haciendo un «barrido» hacia arriba, hasta que toque algún punto; este punto debe pertenecer al cerco convexo. A continuación, tomando como pivote este punto, se continúa «barriendo» hasta encontrar el siguiente punto, y así sucesivamente hasta que el conjunto quede «envuelto» por completo (se vuelva al punto inicial). La Figura 25.2 muestra cómo se descubre el cerco del conjunto de puntos del ejemplo siguiendo este método. El punto B posee la menor coordenada y y se toma como punto inicial. A continuación, M es el primer punto alcanzado por la línea de barido, luego se alcanza L, etcétera.

Por supuesto, realmente no es necesario hacer un barrido para todos los ángulos posibles; solamente se realiza un cálculo estándar para conocer el menor ángulo necesario para encontrar el punto que se alcanzará a continuación. Para cada punto que se incluye en el cerco, se necesita examinar todos los puntos que aún no pertenecen a dicho cerco. Por tanto, este método es bastante pare-

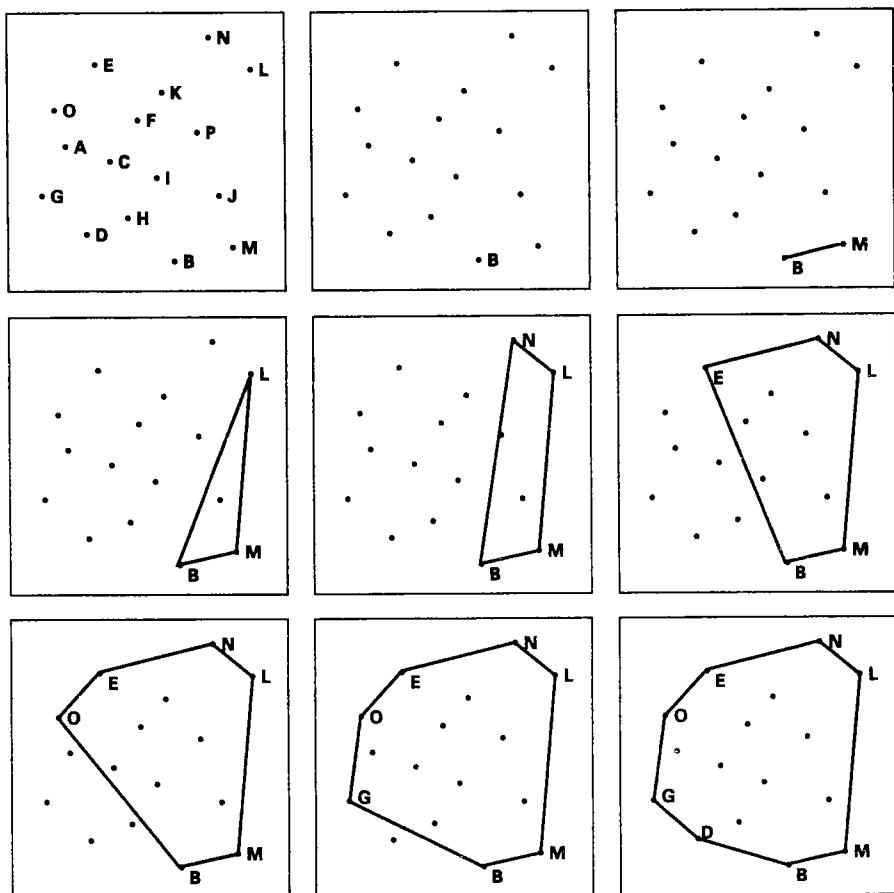


Figura 25.2 Envolventes.

cido al de la ordenación por selección —se elige el «mejor» de los puntos aún no seleccionados, utilizando una búsqueda exhaustiva del mínimo—. En la Figura 25.3 se muestra el movimiento real de datos que se lleva a cabo: la línea M-ésima de la tabla muestra la situación tras incluir el punto M-ésimo en el cerco.

El siguiente programa busca el cerco convexo de un array p de N puntos, representado según la descripción del principio del Capítulo 24. La base de esta implantación es la función theta , desarrollada en el capítulo anterior, que toma dos puntos p_1 y p_2 como argumentos y que se puede considerar que devuelve el ángulo que forma el segmento que une dichos puntos con la horizontal (aunque en realidad devuelve un número más fácil de calcular y que posee las mismas propiedades de ordenación). Por lo demás, la implantación sigue directamente el método antes explicado. Se necesita un centinela para el cálculo del

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
B	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P
B	M	C	D	E	F	G	H	I	J	K	L	A	N	O	P
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P
B	M	L	N	E	O	G	D	I	J	K	C	A	H	F	P

Figura 25.3 Movimiento de datos en envolventes.

ángulo mínimo: aunque normalmente se intentaría disponer las cosas de forma que se utilizase $p[0]$, en este caso es más conveniente utilizar $p[N+1]$.

```
int envolver(punto p[1], int N)
{
    int i, min, M;
    float th, v;
    for (min=0, i=1; i<N; i++)
        if (p[i].y < p[min].y) min = i;
    p[N] = p[min]; th= 0.0;
    for (M=0; M<N; M++)
    {
        intercambio(p, M, min);
        min= N; v= th; th= 360.0;
        for (i=M+1; i<=N; i++)
            if (theta(p[M], p[i]) > v)
                if (theta(p[M], p[i]) < th)
                    { min= i; th= theta(p[M], p[min]); }
        if (min == N) return M;
    }
}
```

En primer lugar, se busca el punto que tiene la menor coordenada y , y se copia en $p[N+1]$ con el fin de detener el bucle, tal como se describe a continuación.

La variable M guarda el número de puntos que se han incluido en el cerco hasta el momento, y v es el valor actual del ángulo de «barrido» (el ángulo que forman la horizontal con la línea que une $p[M-1]$ y $p[M]$). El bucle `for` incluye el último punto encontrado en el cerco intercambiándolo con el punto M -ésimo, y utiliza la función `theta` del capítulo anterior para calcular el ángulo que forman la horizontal y la línea que une dicho punto con cada uno de los puntos que aún no pertenecen al cerco, buscando el punto cuyo ángulo sea el menor de todos los calculados y que al mismo tiempo supere el valor de v . El bucle finaliza cuando se encuentra de nuevo el primer punto (en realidad se trata de la copia del primer punto que se guarda en $p[N+1]$).

Este programa puede o no devolver puntos que se encuentren en un lado del cerco convexo. Esta situación se produce cuando más de un punto tiene el mismo valor de `theta` con $p[M]$ durante la ejecución del algoritmo. Esta implantación devuelve el primer punto que se encuentra, incluso aunque pueda haber otros puntos más próximos a $p[M]$. Cuando sea importante encontrar los puntos situados en los lados de los cercos convexos, se puede modificar `theta` de modo que tenga en cuenta la distancia entre los puntos ofrecidos como argumentos y que, cuando dos puntos tengan el mismo ángulo, asigne un valor menor al punto más próximo.

La principal desventaja de las envolventes es que, en el peor caso, cuando todos los puntos están en el cerco convexo, el tiempo de ejecución es proporcional a N^2 (como en la ordenación por selección). Por otra parte, este método posee la atractiva propiedad de que se puede generalizar a tres (o más) dimensiones. El cerco convexo de un conjunto de puntos de un espacio de dimensión k es el menor polígono convexo que los contiene a todos, quedando definido un polígono convexo por una propiedad, según la cual todas las líneas que unen dos puntos interiores deben estar situadas también en el interior. Por ejemplo, el cerco convexo de un conjunto de puntos en el espacio tridimensional es un objeto tridimensional convexo con caras planas. Se puede calcular «barriendo» el espacio con un plano hasta alcanzar el cerco, y después, «plegado» el plano por las aristas del cerco, tomando como pivotes los diferentes bordes del cerco, hasta que el «paquete» quede «envuelto» (como sucede con numerosos algoritmos geométricos, ¡resulta bastante más sencillo explicar esta generalización que implantarla!).

La exploración de Graham

El siguiente método que se va a examinar, inventado por R.L. Graham en 1972, es interesante porque la mayor parte de los cálculos necesarios se realizan para ordenar: el algoritmo incluye una ordenación, seguida por cálculos relativamente poco costosos (aunque tampoco son fáciles). Utilizando el método del capítulo anterior, el algoritmo comienza construyendo un polígono cerrado simple con los puntos: ordena los puntos utilizando como claves los valores de

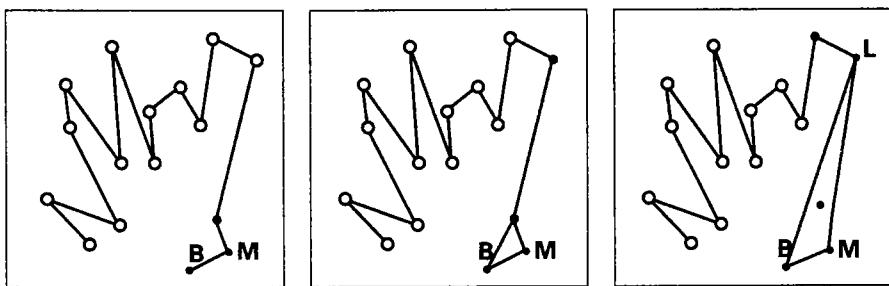


Figura 25.4 Comienzo de la exploración de Graham.

la función theta, correspondientes al ángulo que forman la horizontal y cada una de las líneas que unen los puntos con el pivote $p[1]$ (el punto que tiene la menor coordenada y), de forma que al unir $p[1], p[2], p[3], \dots, p[N], p[1]$ se obtiene un polígono cerrado. En el caso del conjunto de puntos del ejemplo, el resultado es el polígono cerrado simple obtenido en el capítulo anterior. Puede notarse que $p[N], p[1]$ y $p[2]$ son puntos consecutivos del cerco; al ordenarlos, esencialmente se está ejecutando la primera iteración del procedimiento de envolventes (en ambas direcciones).

El cálculo del cerco convexo se completa intentando situar cada punto en el cerco, y eliminando los puntos ya situados que posiblemente no puedan estar en el cerco. En el ejemplo se considera que los puntos tienen el orden B M J L N P K F I E C O A H G D; los primeros pasos se muestran en la Figura 25.4. Al principio, gracias a la ordenación, se sabe que B y M pertenecen al cerco. Cuando se encuentra el punto J, el algoritmo lo incluye en el cerco de prueba de los tres primeros puntos. Después, cuando se encuentra el punto L, el algoritmo determina que J no puede estar en el cerco (ya que, por ejemplo, está dentro del triángulo BML).

En general, no es difícil comprobar qué puntos se deben eliminar. Después de incluir cada punto, se supone que se han eliminado suficientes puntos, de modo que lo trazado hasta el momento podría ser parte del cerco convexo considerando los puntos ya vistos. Conforme se va trazando el cerco, se espera girar a la izquierda de cada vértice del cerco. Si un nuevo punto hace que se gire hacia la *derecha*, entonces el punto recién incluido debe ser eliminado, puesto que existe un polígono convexo que lo contiene. Específicamente, la prueba para eliminar un punto utiliza el procedimiento ccw del capítulo anterior, de la siguiente forma. Suponiendo que se ha determinado que $p[1], \dots, p[M]$ están en el cerco parcial calculado a partir de los puntos $p[1], \dots, p[i-1]$, cuando se tiene que examinar un nuevo punto $p[i]$, se elimina $p[M]$ del cerco si $ccw(p[M], p[M-1], p[i])$ no es negativo. En caso contrario, $p[M]$ aún podría pertenecer al cerco, por lo que no se elimina.

La Figura 25.5 muestra la realización de este proceso utilizando el conjunto de puntos del ejemplo. Conforme se encuentra cada nuevo punto, la situación

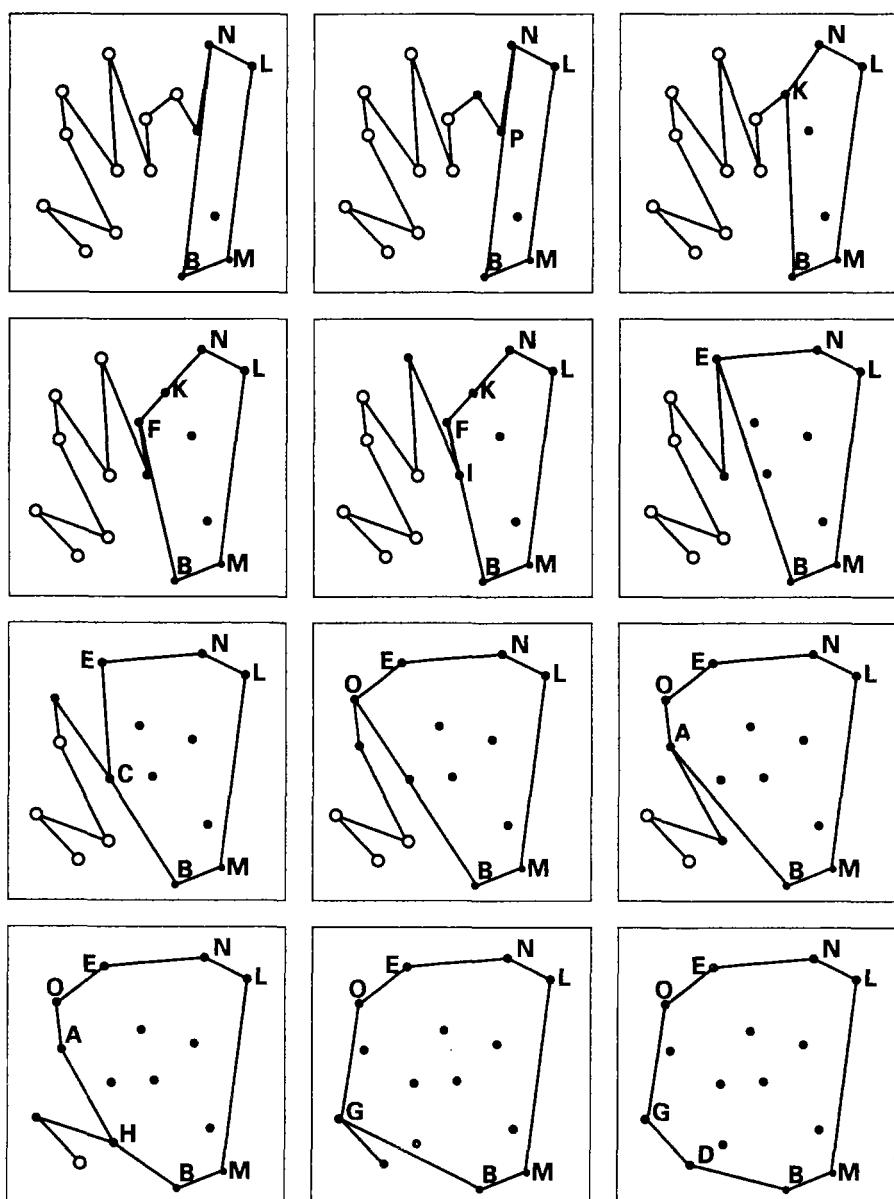


Figura 25.5 Conclusión de la exploración de Graham.

es, en resumen, la siguiente: cada nuevo punto se incluye en el cerco parcial construido hasta el momento, y se utiliza como «testigo» para la eliminación de (cero o más) puntos previamente considerados. Después de incluir L, N y P en el cerco, se elimina P al tener en cuenta el punto K (ya que NPK es un giro a la derecha); después se incluyen F e I, llegando a la consideración del punto E. Llegados a este punto, se debe eliminar I porque FIE es un giro hacia la derecha; F y K se deben eliminar, ya que KFE y NKE son tambien giros hacia la derecha. Por tanto, se puede eliminar más de un punto en el proceso de «vuelta atrás», quizá varios. Continuando de esta forma, el algoritmo vuelve finalmente al punto B.

La ordenación inicial garantiza que cada punto, llegado su turno, se considera como punto del cerco, ya que todos los puntos antes considerados tienen un valor más pequeño de theta. Cada línea que sobrevive a las «eliminaciones» posee la propiedad de que todos los puntos considerados hasta el momento están en el mismo lado, de forma que cuando se vuelve a p[N], que tambien pertenece al cerco debido a la ordenación, se habrá completado el cerco convexo de todos los puntos.

Como en el método de las envolventes, se pueden incluir o no los puntos situados en un lado del cerco, aunque cuando haya puntos colineales se pueden presentar dos situaciones distintas. En primer lugar, si existen dos puntos colineales con p[1], entonces, como antes, la ordenación que hace uso de theta puede ordenarlas o no a lo largo de la línea común. En esta situación, los puntos desordenados serán eliminados durante la exploración. En segundo lugar, se pueden presentar puntos colineales a lo largo del cerco de prueba (que no se eliminan).

Una vez entendido el método básico, su implantación es sencilla, aunque se debe prestar atención a unos cuantos detalles. En primer lugar, el punto que tenga el mayor valor de x de entre todos los puntos que tengan el mínimo valor de y se intercambia con p[1]. A continuación, se utiliza `ordenshell` para reordenar los puntos (también valdría cualquier rutina de ordenación basada en comparaciones), estando implantada la estructura punto como una clase que posee una operación de comparación que compara dos puntos utilizando sus valores de theta con p[1]. Tras la ordenación, se copia p[N] en p[0] para servir como centinela en caso de que p[3] no esté en el cerco. Finalmente, se realiza la exploración antes descrita. El siguiente programa halla el cerco convexo del conjunto de puntos p[1], ..., p[N]:

```
int explgraham(punto p[], int N)
{
    int i, min, M;
    for (min= 1, i= 2; i<= N; i++)
        if (p[i].y < p[min].y) min= i;
    for (i= 1; i<= N; i++)
        if (p[i].y == p[min].y)
```

```

        if (p[i].x > p[min].x) min= i;
        intercambio(p, 1, min);
        ordenshell(p, N);
        p[0] = p[N];
        for (M= 3, i= 4; i<= N; i++)
        {
            while (ccw(p[M], p[M-1], p[i]) >= 0) M--;
            M++; intercambio(p, i, M);
        }
        return M;
    }
}

```

El bucle mantiene un cerco parcial en $p[1], \dots, p[M]$, como se vio anteriormente. Para cada nuevo valor de i considerado, se decrementa M si es necesario, con el fin de eliminar puntos del cerco parcial, y después se intercambia $p[i]$ con $p[M+1]$ para incluirlo (provisionalmente) en el cerco. La Figura 25.6 mues-

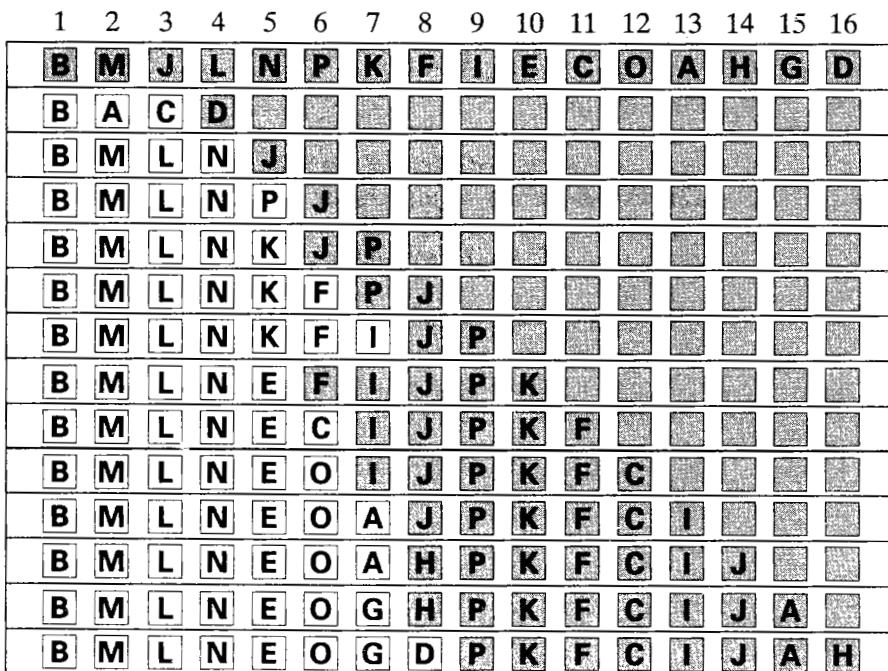


Figura 25.6 Movimiento de datos en la exploración de Graham.

tra el contenido del array p cada vez que se analiza un nuevo punto de este ejemplo.

El lector, si lo desea, puede comprobar por qué para calcular el valor de \min es necesario hallar el punto que tiene la menor coordenada x de entre todos los que tienen la menor coordenada y . Como ya se ha mencionado, otro aspecto sutil a considerar es el hecho de que los puntos colineales poseen el mismo valor de θ , y que es posible que no estén ordenados en el orden en que aparecen en la línea, como se podría esperar.

Una razón por la que resulta interesante estudiar este método es porque se trata de una forma sencilla del *método de retroceso*, una técnica de diseño de algoritmos que se puede resumir como: «intenta algo, y, si no funciona, intenta otra cosa», y que verá de nuevo en el Capítulo 44.

Eliminación interior

Casi todos los métodos de cerco convexo se pueden mejorar enormemente utilizando una sencilla técnica que desecha rápidamente la mayoría de los puntos. La idea general es simple: se cogen cuatro puntos que se sepa que pertenecen al cerco, y se eliminan todos los puntos situados en el interior del cuadrilátero formado por esos cuatro puntos. Esto deja muchos menos puntos a tener en cuenta en, por ejemplo, la exploración de Graham o en la técnica de las envolventes.

Los cuatro puntos que se sabe que pertenecen al cerco se deberían elegir teniendo en cuenta cualquier información disponible acerca de los puntos de entrada. En general, es mejor adaptar la elección de los puntos a la distribución de la entrada. Por ejemplo, si todos los valores de x e y , dentro de determinados límites, son igualmente probables (una distribución rectangular), al elegir cuatro puntos de las esquinas (los cuatro puntos que tienen la mayor y menor suma y diferencia de sus coordenadas) se eliminan casi todos los puntos. La Figura 25.7 muestra que esta técnica elimina la mayoría de los puntos que no están en el cerco de los dos conjuntos de puntos del ejemplo.

En una implantación del método de la eliminación interior, el «bucle interior» para los conjuntos de puntos aleatorios es el que comprueba si un punto está situado dentro del cuadrilátero de prueba. Esto se puede acelerar algo utilizando un rectángulo cuyos lados sean paralelos a los ejes x e y . A partir de las cuatro coordenadas que definen el cuadrilátero, es fácil calcular el rectángulo más grande que cabe en dicho cuadrilátero. Si se utiliza este rectángulo, se eliminarán menos puntos del interior, pero la velocidad de la comprobación compensa con creces esta pérdida.

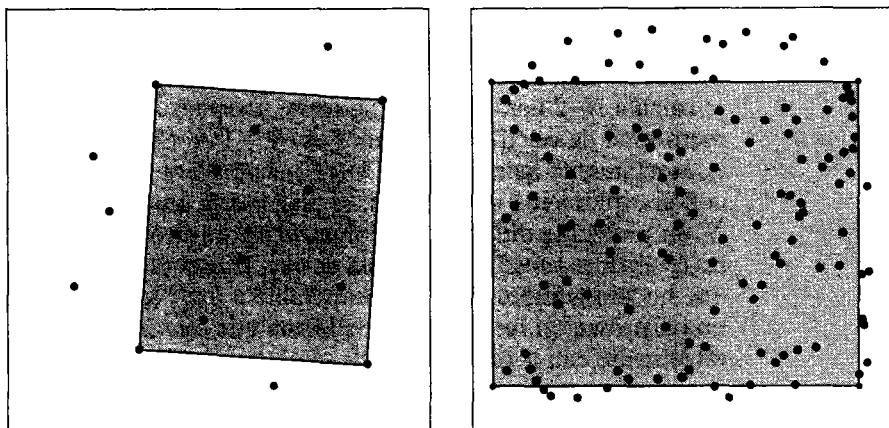


Figura 25.7 Eliminación interior.

Rendimiento

Como se dijo en el capítulo anterior, los algoritmos geométricos son algo más difíciles de analizar que los algoritmos de algunas otras áreas que se han estudiado, ya que la entrada (y la salida) es más difícil de caracterizar. A menudo no tiene sentido hablar de conjuntos de puntos «aleatorios»: por ejemplo, conforme crece N , el cerco convexo de los puntos de una distribución rectangular es muy probable que esté muy próximo al rectángulo que define dicha distribución. Los algoritmos que se han visto dependen de diferentes propiedades de la distribución del conjunto de puntos, y, por ello, en la práctica no son comparables, pues para compararlos analíticamente se necesitaría entender unas interacciones muy complicadas entre las propiedades (poco conocidas) de los conjuntos de puntos. Por otra parte, se pueden decir algunas cosas sobre el rendimiento de los algoritmos que pueden ayudar a la hora de elegir uno de ellos para una aplicación concreta.

Propiedad 25.1 *Después de la ordenación, la exploración de Graham es un proceso lineal en el tiempo.*

Es necesario reflexionar un momento para convencerse uno mismo de que esto es cierto, ya que en el programa hay un «bucle dentro de otro bucle». Sin embargo, es fácil ver que ningún punto se «elimina» más de una vez, por lo que dentro del doble bucle, el código itera menos de N veces. Utilizando este método, el tiempo total necesario para hallar el cerco convexo está en $O(N \log N)$, pero el «bucle interior» del método es la propia ordenación, que se puede hacer más eficaz utilizando las técnicas de los Capítulos 8 a 12.■

Propiedad 25.2 *Si hay M vértices en el cerco, la técnica de las envolventes necesita unos MN pasos*

En primer lugar, hay que calcular $N-1$ ángulos para hallar el mínimo, después se calcula $N-2$ para hallar el siguiente, después $N-3$, etc., de modo que el número total de cálculos de ángulos es $(N - 1) + (N - 2) + \dots + (N - M + 1)$, que es exactamente igual a $MN - M(M - 1)/2$. Para comparar analíticamente esto con la exploración de Graham, se necesitaría una fórmula de M en función de N , un problema difícil en la geometría estocástica. Para una distribución circular (y algunas otras), la respuesta es que M está en $O(N^{1/3})$, y para valores de N que no son grandes, $N^{1/3}$ es comparable a $\log N$ (que es el valor esperado para una distribución rectangular), de forma que este método competiría muy favorablemente con la exploración de Graham. Por supuesto, siempre se debe tener en cuenta el peor caso N^2 . ■

Propiedad 25.3 *El método de la eliminación interior es, por término medio, lineal.*

El análisis matemático completo de este método requeriría una geometría estocástica incluso más sofisticada que antes, pero el resultado general es el que indica la intuición: casi todos los puntos están dentro del cuadrilátero y se descartan —el número de puntos que quedan está en $O(\sqrt{N})$ —. Esto es cierto incluso si se utiliza el rectángulo, como se mencionó anteriormente. Esto hace que el tiempo medio de ejecución de todo el algoritmo de cerco convexo sea proporcional a N , ya que la mayoría de los puntos sólo se examinan una vez (cuando se descartan). Por regla general, no importa mucho qué método se utiliza después, ya que es probable que queden muy pocos puntos. No obstante, para defenderse ante el peor caso (cuando todos los puntos están en el cerco), es prudente utilizar la exploración de Graham. Con esto se consigue un algoritmo que es casi seguro que se ejecutará en la práctica de forma lineal en el tiempo, y que se garantiza que se ejecutará en un tiempo proporcional a $N\log N$. ■

El resultado del caso medio de la propiedad 25.3 sólo es válido para puntos distribuidos aleatoriamente en un rectángulo, y en el peor caso, el método de eliminación interior no elimina nada. No obstante, para otras distribuciones u otros conjuntos de puntos de propiedades desconocidas, aún se recomienda utilizar este método porque su coste es bajo (una exploración lineal de los puntos, con unas pocas comprobaciones), y el ahorro posible es alto (la mayoría de los puntos se pueden eliminar fácilmente). El método también se puede ampliar a dimensiones mayores.

Es posible concebir una versión recursiva del método de eliminación interior: se hallan los puntos extremos, y se eliminan los puntos situados en el interior del cuadrilátero definido, como antes, pero considerando después que los puntos restantes se dividen en subproblemas que se pueden resolver de forma independiente, utilizando el mismo método. Esta técnica recursiva es similar al

procedimiento de selección `selecc` de tipo Quicksort que se vio en el Capítulo 9. Como aquel procedimiento, es vulnerable a un tiempo de ejecución N^2 en el peor caso. Por ejemplo, si todos los puntos originales están en el cerco convexo, no se desecha ningún punto en la etapa recursiva. Como en `selecc`, el tiempo de ejecución, por término medio, es lineal (aunque no resulta fácil demostrarlo). Pero debido a que se eliminan tantos puntos en la primera etapa, no es probable que merezca la pena preocuparse por realizar una posterior descomposición recursiva en ninguna aplicación práctica.

Ejercicios

1. Suponiendo que se conoce de antemano que el cerco convexo de un conjunto de puntos es un triángulo, obtener un algoritmo sencillo que encuentre dicho triángulo. Responder a la misma cuestión en el caso de que el cerco sea un cuadrilátero.
2. Indicar un método eficaz para determinar si un punto está situado en el interior de un polígono convexo.
3. Implantar un algoritmo de cerco convexo parecido a la inserción ordenada, utilizando el método del ejercicio anterior.
4. En la exploración de Graham ¿es estrictamente necesario empezar con un punto que con seguridad pertenece al cerco? Explicar las razones.
5. En el método de la envolvente ¿es estrictamente necesario empezar con un punto que con seguridad pertenece al cerco? Explicar las razones.
6. Dibujar un conjunto de puntos que haga que la exploración de Graham del cerco convexo sea particularmente ineficaz.
7. ¿Es capaz la exploración de Graham de encontrar el cerco convexo de los puntos que constituyen los vértices de *cualquier* polígono sencillo? Explicar por qué, o buscar un contraejemplo que demuestre lo contrario.
8. ¿Qué cuatro puntos se deberían utilizar en el método de la eliminación interior si se supone que la entrada está distribuida aleatoriamente en el interior de una circunferencia (utilizando coordenadas polares aleatorias)?
9. Comparar empíricamente la exploración de Graham y el método de la envolvente para conjuntos de puntos grandes en los que los valores de x e y son equiprobables dentro del intervalo 0 a 1.000.
10. Implantar el método de la eliminación interior y determinar empíricamente cómo debería ser el valor de N antes de que se pueda esperar que queden 50 puntos tras utilizar el método en conjuntos de puntos en los que los valores de x e y son equiprobables dentro del intervalo 0 a 1.000.

Búsqueda por rango

Dado un conjunto de puntos del plano, es natural preguntar cuáles de ellos se encuentran dentro de una zona específica. «Listar todas las ciudades que estén a menos de 50 millas de Princeton» es una pregunta que lógicamente podría hacerse si se dispusiera del conjunto de puntos correspondientes a las ciudades de los Estados Unidos. Cuando se limitan las figuras geométricas a los rectángulos, el problema se extiende fácilmente a dominios no geométricos. Por ejemplo, «listar todas las personas entre 21 y 25 años con ingresos entre 60.000 y 100.000 dólares» es lo mismo que preguntar qué «puntos» de un archivo de datos con nombres de personas, edades e ingresos, están dentro de un cierto rectángulo del plano edad-ingresos.

La generalización a más de dos dimensiones es inmediata. Si se desea listar todas las estrellas a menos de 50 años luz del sol, se tiene un problema tridimensional, y si se desea conocer del conjunto de personas jóvenes y bien pagadas del párrafo anterior las que son altas y mujeres, se tiene un problema de cuatro dimensiones. De hecho, la dimensión de tales problemas puede llegar a ser muy grande.

En general, se supone la existencia de un conjunto de *registros* con ciertos *atributos* que toman valores en un conjunto ordenado. (Esto a veces se denombra una *base de datos*, aunque para este término se han desarrollado definiciones más específicas y completas.) A la acción de encontrar todos los registros de una base de datos de los que un conjunto específico de atributos satisfacen determinadas restricciones de rango, se la denomina *búsqueda por rango*, y es un problema difícil e importante en ciertas aplicaciones prácticas. En este capítulo se centrará la atención en el problema geométrico bidimensional en el que los registros son puntos y los atributos sus coordenadas, para posteriormente presentar otras posibles generalizaciones.

Los métodos que se estudiarán son generalizaciones directas de las técnicas que ya se han visto en la búsqueda sobre claves simples (en una dimensión). Se supone que gran parte de las consultas se hacen sobre el mismo conjunto de puntos, lo que permite dividir el problema en dos partes: un algoritmo de *pre-*

procesamiento, que estructure los puntos dados para permitir una búsqueda por rango eficaz, y un algoritmo de *búsqueda por rango* que utilice dicha estructura para devolver los puntos situados dentro de cualquier rango (multidimensional). Esta separación hace difícil la comparación de métodos diferentes, puesto que el coste total depende no sólo de la distribución de los puntos implicados sino también del número y la naturaleza de las peticiones.

El problema de la búsqueda por rango en una dimensión consiste en devolver todos los puntos que están dentro de un intervalo específico. Esto se puede hacer ordenando los puntos en preprocessamiento y haciendo luego una búsqueda binaria sobre los puntos extremos del intervalo para devolver todos los puntos que estén entre ellos. Otra solución consiste en construir un árbol binario de búsqueda y después hacer un simple recorrido recursivo del mismo, devolviendo los puntos del intervalo e ignorando las partes del árbol situadas fuera de él. El programa que se necesita es un simple recorrido recursivo del árbol (ver los Capítulos 4 y 14). Si el punto del extremo izquierdo del intervalo está a la izquierda del punto de la raíz, se busca (recursivamente) en el subárbol izquierdo y de forma similar para el derecho, verificando en cada nodo que se encuentre si los puntos asociados a cada uno de ellos están o no dentro del intervalo:

```
int rango(int v1, int v2)
    { return rangol(cabeza->der, v1, v2); }
int rangol(struct nodo *t, int v1, int v2)
{
    int tx1, tx2, contador = 0;
    if (t == z) return 0;
    tx1 = (t->clave >= v1);
    tx2 = (t->clave <= v2);
    if (tx1) contador += rangol(t->izq, v1, v2);
    if (tx1 && tx2) contador++;
    if (tx2) contador += rangol(t->der, v1, v2);
    return contador;
}
```

Dependiendo del contexto, estas operaciones pudieran ser mejoras de la implementación del diccionario del árbol binario de búsqueda del Capítulo 14. La Figura 26.1 muestra los puntos que se encuentran cuando este programa se ejecuta sobre un árbol de prueba. Se observa que los puntos devueltos no necesitan estar enlazados al árbol.

Propiedad 26.1 Una búsqueda por rango unidimensional se puede hacer con un número de pasos en $O(N \log N)$ para el preprocessamiento y en $O(R + \log N)$ para la búsqueda por rango, donde R es el número de puntos que están realmente en el intervalo.

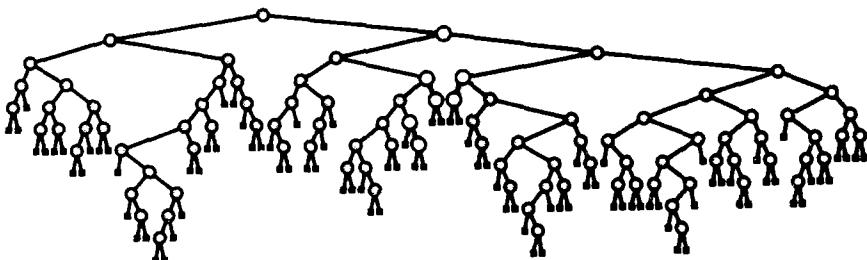


Figura 26.1 Búsqueda por rango (unidimensional) con un árbol de búsqueda binaria.

Esto se obtiene directamente de las propiedades elementales de las estructuras de búsqueda (ver Capítulos 14 y 15). Se podría utilizar, si se desea, un árbol equilibrado.■

El objetivo de este capítulo será alcanzar los mismos tiempos de ejecución para la búsqueda por rango multidimensional. El parámetro R puede ser bastante significativo: dada la facilidad para hacer consultas de rango, un usuario podría fácilmente formular peticiones que impliquen a todos o casi todos los puntos. Este tipo de petición podría ocurrir razonablemente en muchas aplicaciones, pero no se necesitan algoritmos sofisticados si *todas* las peticiones son de este tipo. Los algoritmos que se van a considerar se han diseñado para ser eficaces en peticiones donde no se espera que se devuelva un gran número de puntos.

Métodos elementales

En dos dimensiones, el «rango» es una zona del plano. Por simplicidad, se considerará el problema de encontrar todos los puntos cuyas coordenadas x estén dentro de un intervalo en x dado y cuyas coordenadas y estén dentro de un intervalo en y : esto es, se buscan todos los puntos que están dentro de un rectángulo dado. Así pues, se supone que existe un tipo `rect` que es un registro de cuatro enteros, los puntos extremos de los intervalos horizontal y vertical. La operación básica consiste en comprobar si un punto dado está dentro de un rectángulo dado, por lo que se supone una función `dentro_rect(struct punto p, struct rect r)` que comprueba esto de forma directa, devolviendo un valor distinto de cero si p está dentro de r . El objetivo es encontrar todos los puntos situados en el interior de un rectángulo dado, utilizando la menor cantidad posible de llamadas a `dentro_rect`.

La forma más simple de resolver este problema es la *búsqueda secuencial*: se recorren todos los puntos, comprobando si cada uno está dentro del rango especificado (llamando a `dentro_rect` para cada uno de ellos). Este método se

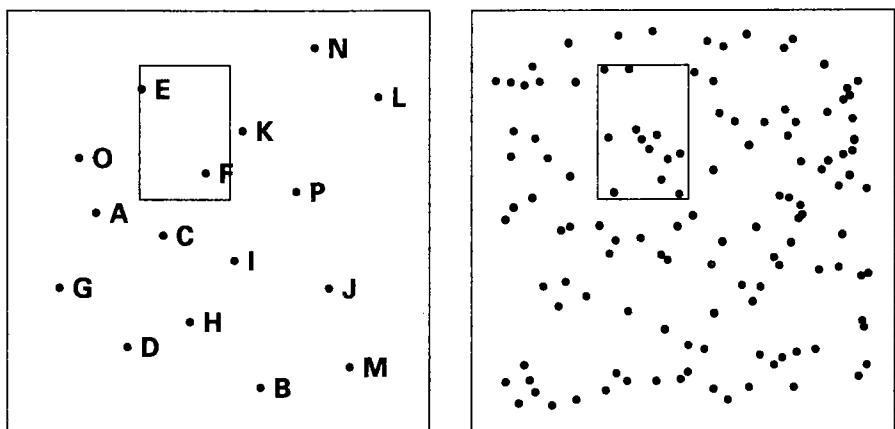


Figura 26.2 Búsqueda por rango bidimensional.

utiliza de hecho en muchas aplicaciones de bases de datos porque se puede mejorar fácilmente «empaquetando» las peticiones, y así se comprueban muchas de ellas durante el propio recorrido a lo largo de los puntos. En una base de datos muy grande, donde éstos se encuentran en dispositivos externos y el tiempo de lectura es el factor de coste dominante, éste puede ser un método muy razonable: reagrupar tantas preguntas como sea posible tener en la memoria interna y buscar la respuesta a todas ellas en una sola pasada a través del gran archivo de datos externo. Sin embargo, si este tipo de empaquetado no es conveniente o si la base de datos es, en cierto sentido, pequeña, existen métodos mucho mejores.

No obstante, en este problema geométrico la búsqueda secuencial parece implicar demasiado trabajo, como se muestra en la Figura 26.2. El rectángulo de búsqueda posiblemente contenga sólo unos pocos puntos, de modo que ¿es necesario buscar a través de todos los puntos para encontrar sólo unos pocos? Una mejora simple del método de búsqueda secuencial consiste en la aplicación directa de un método unidimensional conocido a lo largo de una o más de las dimensiones de la búsqueda. Por ejemplo, se buscan los puntos cuyas coordenadas x estén dentro del intervalo x especificado por el rectángulo, y luego se verifican las coordenadas y de esos puntos para determinar si pueden estar (o no) en el interior del rectángulo. Así pues, los puntos que no pueden estar dentro del rectángulo porque sus coordenadas x están fuera de rango no se examinarán nunca. Esta técnica se denomina *proyección*; por supuesto, también sería posible proyectar sobre y . En el ejemplo, se comprobarían los puntos E, C, H, F e I para la proyección x , y los O, E, F, K, P, N y L para la proyección y . Se observa que el conjunto de puntos buscados (E y F) son precisamente aquellos que aparecen en ambas proyecciones.

Si los puntos están uniformemente distribuidos en una región rectangular,

entonces es fácil calcular el número medio de puntos a verificar. La proporción de puntos que se podría esperar encontrar en un rectángulo dado es simplemente la relación entre su área y la de toda la región; la proporción de puntos que se podría esperar verificar para una proyección x es la relación entre el ancho del rectángulo y el de la región, y lo mismo para una proyección y . En el ejemplo, utilizando un rectángulo de 4 por 6 en una región de 16 por 16 se podría esperar encontrar $3/32$ puntos en el rectángulo, $1/4$ de ellos en una proyección x y $3/8$ en una y . Evidentemente, en tales circunstancias, es mejor proyectar sobre el eje correspondiente a la más pequeña de las dos dimensiones del rectángulo. Por otro lado, es fácil construir casos donde la técnica de proyección podría fallar miserablemente: por ejemplo, si el conjunto de puntos forma una figura en forma de «L» y la búsqueda se efectúa en un rango que engloba sólo la esquina derecha de la «L», entonces la proyección sobre los ejes elimina sólo la mitad de los puntos.

A primera vista, parece que la técnica de proyección podría mejorarse «intersecando» los puntos que están dentro del rango x y los que están dentro del y . Pero intentar hacer esto sin examinar todos los puntos del rango x o todos los del rango y , en el peor caso, es tan difícil que sirve principalmente para que se aprecien mejor los métodos más sofisticados que se van a estudiar.

Método de la rejilla

Una técnica simple pero eficaz para mantener relaciones de proximidad entre los puntos del plano consiste en construir una rejilla imaginaria que divida la zona de búsqueda en pequeñas celdas y en mantener listas de pequeño tamaño de los puntos que están dentro de cada celda. (Ésta es una técnica que se utiliza, por ejemplo, en arqueología.) Así, cuando se buscan los puntos incluidos en un rectángulo dado, sólo se necesita buscar en las listas que corresponden a las celdas del rectángulo. En el ejemplo, sólo se examinan E, C, F y K, como se muestra en la Figura 26.3.

Queda todavía por determinar el tamaño de la rejilla: si es muy grosera, cada celda contendrá demasiados puntos, y, si es muy fina, habrá muchas celdas en las que buscar (aunque la mayoría estén vacías). Una forma de alcanzar el equilibrio entre estos dos extremos es escoger el tamaño de la rejilla de modo que el número de celdas sea una fracción constante del número total de puntos, lo que da un número medio de puntos en cada celda aproximadamente igual a una pequeña constante. Para el pequeño conjunto de puntos del ejemplo, la utilización de una rejilla de 4 por 4, con 16 puntos, significa que cada celda contendrá por término medio un punto.

A continuación se presenta una implementación directa de una clase que soporta la búsqueda por rango en un espacio bidimensional utilizando el método de la rejilla. Las variables `maxR` y `talla` se utilizan para controlar la resolución de la rejilla (el número y tamaño de las celdas). Se supone que las coor-

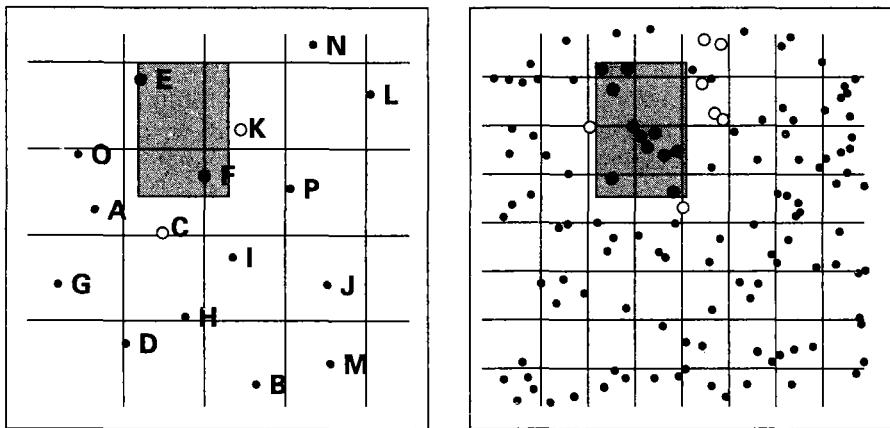


Figura 26.3 Método de la rejilla para la búsqueda por rango.

denadas son enteras, como es habitual, y `talla` se toma como el ancho de la celda. Existen `maxR` por `maxR` celdas en la rejilla de modo que el rango de las coordenadas de los puntos varía entre 0 y `talla*maxR`. Para encontrar la celda a la que pertenece un punto dado, se dividen sus coordenadas por `talla`:

```
class Rango
{
private:
    struct nodo
    {
        struct punto p; struct nodo *siguiente; };
    struct nodo *rejilla[maxR][maxR];
    struct nodo *z;
public:
    Rango();
    void insertar(struct punto p);
    int buscar(rect rango);
};
Rango::Rango()
{
    int i, j;
    z = new nodo;
    for (i = 0; i <= maxR; i++)
        for (j = 0; j <= maxR; j++)
            rejilla[i][j];
}
```

```

void Rango::insertar(struct punto p)
{
    struct nodo *t = new nodo;
    t->p = p; t->siguiente = rejilla[p.x/talla][p.y/talla];
    rejilla[p.x/talla][p.y/talla] = t;
}

```

Este programa utiliza la representación estándar por listas enlazadas, con el nodo cola ficticio z y la lista de los encabezamientos en el array. Las listas están desordenadas, con inserción por el frente, como en el Capítulo 3.

Los valores apropiados de las variables $talla$ y $maxR$ dependen del número de puntos, de la cantidad de memoria disponible y del rango de los valores de las coordenadas. En primera aproximación, si hay N puntos y se desea M puntos por celda, entonces se necesita alrededor de N/M celdas, por lo que se debe escoger $maxR$ como el entero más próximo a $\sqrt{N/M}$ y $talla$ debe ser aproximadamente el máximo de coordenadas de punto dividido por $\sqrt{N/M}$. Esto proporciona alrededor de N/M celdas. Estas estimaciones son falsas para pequeños valores de los parámetros, pero son válidas en la mayoría de los casos, pudiéndose adaptar fácilmente para aplicaciones específicas. No es necesario un cálculo preciso. Típicamente se podría utilizar una potencia de dos para el valor de $talla$ para hacer la multiplicación y la división por $talla$ mucho más eficaz, doblando solamente el número de puntos por celda (de 1 a 2 en el ejemplo anterior).

La implementación precedente utiliza $M = 1$, una opción que se elige con mucha frecuencia. Si no hay problemas de espacio en la memoria, pueden ser apropiados valores más grandes, pero los más pequeños probablemente no serán útiles salvo en situaciones muy específicas.

Ahora, la mayor parte del trabajo de la búsqueda por rango se efectúa simplemente indexando dentro del array $rejilla$, como sigue:

```

int Rango::buscar(struct rect rango) //Método de rejilla
{
    struct nodo *t;
    int i, j, contador = 0;
    for (i = rango.x1/talla; i <= rango.x2/talla; i++)
        for (j = rango.y1/talla; j <= rango.y2/talla; j++)
            for (t = rejilla[i][j]; t != z; t = t->siguiente)
                if (dentro_rect(t->p, rango)) contador++;
    return contador;
}

```

Este programa cuenta simplemente el número de puntos del rango, utilizando una variable global $contador$. Modificarlo para que imprima o devuelva todos los puntos del rango es un proceso directo.

Propiedad 26.2 *El método de la rejilla para la búsqueda por rango es lineal en el número de puntos del rango, por término medio, y lineal en el número total de puntos en el peor caso.*

Esto depende de la elección de los parámetros de modo que el número esperado de puntos en cada celda sea constante, como se describió con anterioridad. Si el número de puntos del rectángulo a buscar es R , entonces el número de celdas de la rejilla a examinar es proporcional a R . El número de celdas examinadas que no están completamente dentro del rectángulo es ciertamente menor que una pequeña constante multiplicada por R , por lo que el tiempo total de ejecución (por término medio) es lineal en R . Para un R grande, el número de puntos examinados no incluidos en el rectángulo buscado es bastante pequeño: todos los puntos de este tipo están en las celdas que intersecan al lado del rectángulo de búsqueda, y el número de celdas con esta propiedad es proporcional a \sqrt{R} , para un R grande. Esta observación es falsa si las celdas son muy pequeñas (muchas celdas vacías dentro del rectángulo) o demasiado grandes (muchos puntos en las celdas del perímetro del rectángulo) o si el rectángulo de búsqueda es más estrecho que una celda (podría intersecar muchas celdas pero tener pocos puntos dentro).■

El método de la rejilla es eficaz si los puntos están bien distribuidos sobre el rango, pero no lo es si están agrupados. (Por ejemplo, todos los puntos podrían estar en una celda, lo que significaría que la estrategia de la rejilla no habría servido para nada.) El método que se examinará a continuación hace que sea muy poco probable este peor caso subdividiendo el espacio de manera no uniforme, adaptándolo a cada conjunto de puntos.

Árboles bidimensionales

Los *árboles bidimensionales (2D)* son estructuras dinámicas y adaptables muy similares a los árboles binarios, pero que dividen el espacio geométrico de una manera apropiada para su utilización en la búsqueda por rango y en otros problemas. La idea es construir árboles binarios de búsqueda con puntos en los nodos, utilizando, en una secuencia estrictamente alternada, las coordenadas x y y de esos puntos como claves.

El mismo algoritmo de inserción en árboles binarios de búsqueda normales se utiliza para insertar puntos en árboles 2D, pero situando en la raíz la coordenada y (si el punto a insertar tiene una y menor que la de la raíz, se va hacia la izquierda; si no a la derecha), después la x en el siguiente nivel, en el siguiente la y , y así sucesivamente, alternando hasta que se encuentre un nodo externo. La Figura 26.4 muestra el árbol 2D correspondiente al pequeño ejemplo del conjunto de puntos.

La importancia de esta técnica es que corresponde a dividir el plano de una

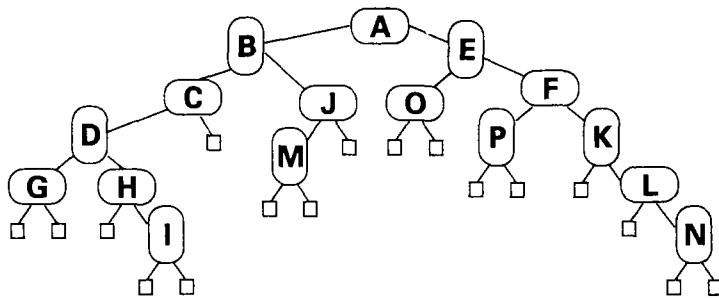


Figura 26.4 Un árbol bidimensional (2D).

forma simple: todos los puntos por debajo del de la raíz van al subárbol izquierdo y todos los que están por encima van al subárbol derecho; después todos los puntos por encima del de la raíz y a la izquierda del punto del subárbol derecho van al subárbol izquierdo del subárbol derecho de la raíz, etcétera.

Las Figuras 26.5 y 26.6 muestran cómo se subdivide el plano en correspondencia con la construcción del árbol de la Figura 26.4. Primero se dibuja una línea horizontal que pase por A, el primer nodo insertado. Después, como B está por debajo de A, va a la izquierda de A en el árbol y se divide el semiplano que queda por debajo de A por medio de una línea vertical que pasa por la coordenada x de B (segundo diagrama de la Figura 26.5). A continuación, puesto que C está por debajo de A, se va a la izquierda de la raíz, y como está a la izquierda de B se va a la izquierda de B, dividiendo la porción del plano por debajo de A y a la izquierda de B por medio de una línea horizontal que pase por la coordenada y de C (tercer diagrama de la Figura 26.5). La inserción de D es similar, después E va a la derecha de A, puesto que está por encima (primer diagrama de la Figura 26.6), etcétera.

Cada nodo externo del árbol corresponde a un rectángulo del plano. Cada región corresponde a un nodo externo del árbol; cada punto pertenece a un seg-

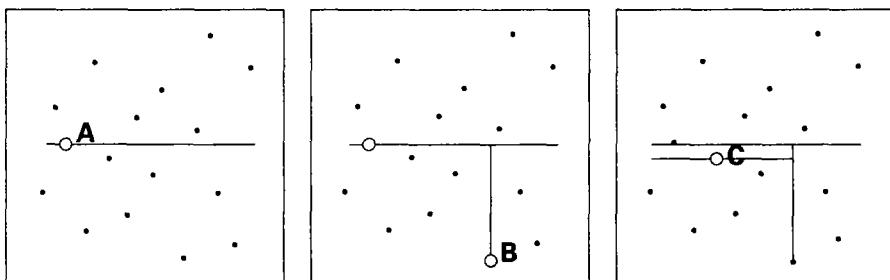


Figura 26.5 Subdivisión del plano con un árbol 2D: etapas iniciales.

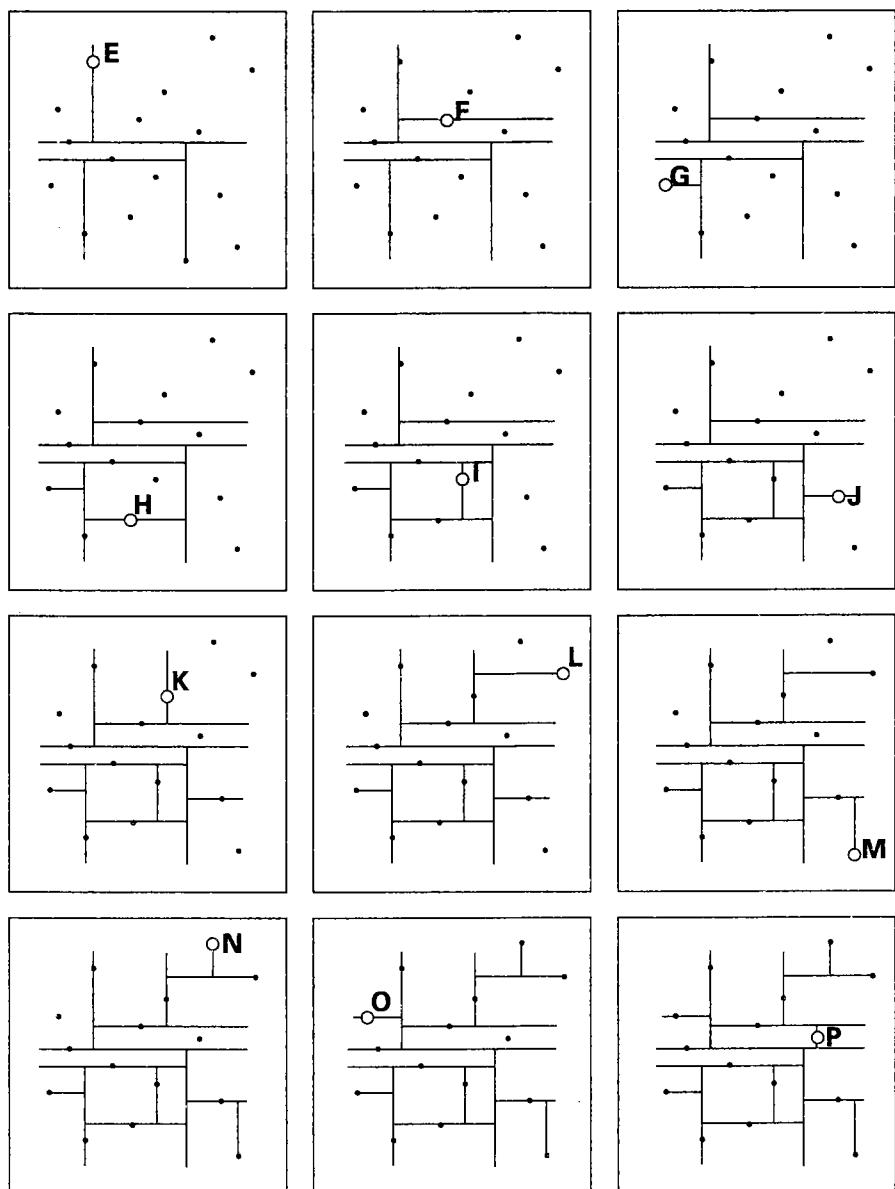


Figura 26.6 Subdivisión del plano con un árbol 2D: continuación.

mento horizontal o vertical que define la división efectuada en el árbol en ese punto.

El código para la construcción de árboles 2D es una modificación directa de la búsqueda-inserción estándar en un árbol binario que permita estar cambiando entre las coordenadas x e y en cada nivel:

```
class Rango
{
private:
    struct nodo
    {
        struct punto p; struct nodo *izq, *der;
    };
    struct nodo *z, *cabeza;
    struct punto ficticio;
    int buscar_rect(struct nodo *t,
                    struct rect rango, int d);
public:
    Rango();
    void insertar(struct punto p);
    int buscar(rect rango);
};

void Rango::insertar(struct punto p) //Arbol 2D
{
    struct nodo *f, *t; int d, td;
    for (d = 0, t = cabeza; t != z; d != ! d)†
    {
        td = d ? (p.x < t->p.x) : (p.y < t->p.y);
        f = t; t = td ? t->izq : t->der;
    }
    t = new nodo; t->p = p; t->izq = z; t->der = z;
    if (td) f->izq = t; else f->der = t;
}
```

La interfaz pública para esta clase es la misma que para el método de la rejilla anterior, pero la implementación utiliza árboles binarios de búsqueda.

Propiedad 26.3 *La construcción de un árbol 2D para N puntos aleatorios necesita por término medio $2N\log N$ comparaciones.*

En efecto, para puntos aleatoriamente distribuidos, los árboles 2D tienen las

[†] En el original $!=$. Pensamos que d debe ser una variable lógica o booleana, ya que, en función de su valor (verdadero o falso), se irá alternativamente al subárbol izquierdo o al subárbol derecho. En otros libros del autor, *Algorithms* y *Algorithms in C*, figura expresamente d como variable lógica. (*N. de los T.*)

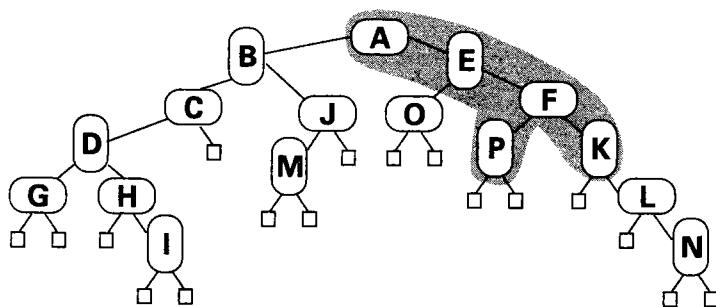


Figura 26.7 Búsqueda por rango con un árbol 2D.

mismas características de rendimiento que los árboles binarios de búsqueda. Las dos coordenadas actúan como «claves» aleatorias.■

Para efectuar una búsqueda por rango utilizando árboles 2D, primero se construye el árbol 2D insertando N puntos en un árbol inicialmente vacío. El código de inicialización debe estar cuidadosamente coordinado con las condiciones iniciales del procedimiento de descenso por el árbol, pues si no podría introducirse un error inoportuno, y el algoritmo buscaría coordenadas x donde el árbol tiene las y , y viceversa.

A continuación, para efectuar la búsqueda por rango, se compara el punto de cada nodo con el rango de la dimensión utilizada para dividir el plano en ese nodo. Para el ejemplo, se comienza por ir a la derecha de la raíz y a la derecha del nodo E, puesto que el rectángulo está enteramente por encima de A y a la derecha de E. Después, en el nodo F, se debe descender por ambos subárboles, puesto que F está dentro del rango x definido por el rectángulo (lo que *no* equivale a decir que F está dentro del rectángulo). Luego se comprueban los subárboles izquierdos de P y K, lo que corresponde a verificar que regiones del plano se solapan con el rectángulo de búsqueda. (Ver las Figuras 26.7 y 26.8.)

Este proceso se implementa fácilmente con una generalización directa del procedimiento rango de una dimensión (1D) que se estudió al principio de este capítulo:

```

int Rango::buscar(struct rect rango) // árbol 2D
{
    return buscar_rect(cabeza->der, rango, 1);
}
int Rango::buscar_rect(struct nodo *t, struct rect rango, int d)
{
    int t1, t2, tx1, tx2, ty1, ty2, contador = 0;
    if (t == z) return 0;
    tx1 = rango.x1 < t->p.x; tx2 = t->p.x <= rango.x2;
    ty1 = rango.y1 < t->p.y; ty2 = t->p.y <= rango.y2;
    t1 = d ? tx1 : ty1; t2 = d ? tx2 : ty2;
```

```

if (t1) contador += buscar_rect(t->izq, rango, !d);
if (dentro_rect(t->p, rango)) contador++;
if (t2) contador += buscar_rect(t->der, rango, !d);
return contador;
}

```

Este procedimiento desciende por los dos subárboles sólo cuando la línea de división corta al rectángulo, lo que no debería pasar frecuentemente para rectángulos relativamente pequeños. La Figura 26.8 muestra la subdivisión del plano y los puntos examinados en los dos ejemplos.

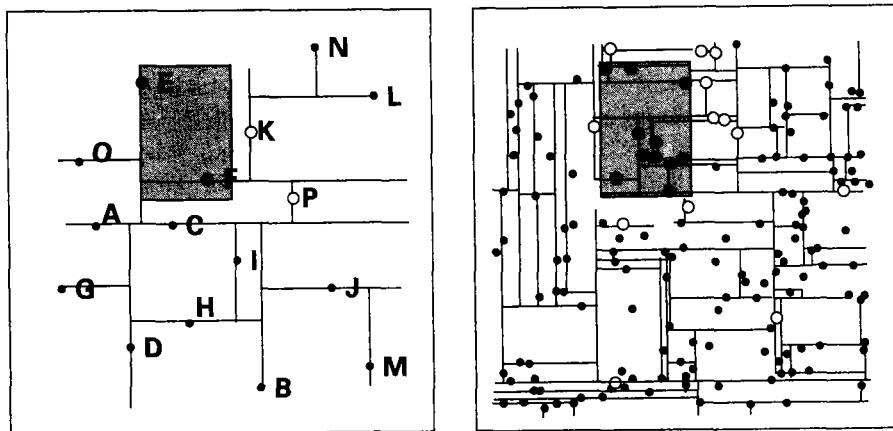


Figura 26.8 Búsqueda por rango con un árbol 2D (subdivisión del plano).

Propiedad 26.4 *La búsqueda por rango con un árbol 2D parece utilizar alrededor de $R + \log N$ pasos para encontrar R puntos que están en rangos de tamaño razonable en una región que contiene N puntos.*

El análisis de este método está todavía por hacer y la propiedad planteada es una conjectura basada en la evidencia empírica. Por supuesto, el rendimiento (y el análisis) depende mucho del tipo de rango utilizado. Pero el método es comparable en su rendimiento con el de la rejilla y, en cierto modo, depende menos de la «aleatoriedad» del conjunto de puntos. La Figura 26.9 muestra el árbol 2D del ejemplo mayor.■

Búsqueda por rango multidimensional

El método de la rejilla y el de los árboles 2D se generalizan de forma directa para más de dos dimensiones: las extensiones simples y directas de los algorit-

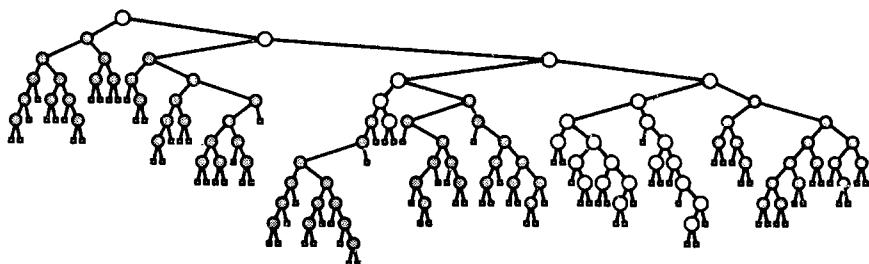


Figura 26.9 Búsqueda por rango con un árbol 2D grande.

mos anteriores proporcionan métodos de búsqueda por rango para más de dos dimensiones. Sin embargo, la naturaleza del espacio multidimensional impone cierta precaución y sugiere que las características de rendimiento de los algoritmos pueden ser difíciles de predecir para una aplicación particular.

Para implementar el método de la rejilla para búsquedas k -dimensionales, se toma simplemente un array rejilla k -dimensional con un índice para cada dimensión. El problema principal consiste en elegir un valor razonable para talla. Este problema resulta bastante obvio cuando se considera un k grande: ¿qué tipo de rejilla se debe utilizar para una búsqueda 10-dimensional? Incluso si se utilizan sólo tres divisiones por dimensión, se necesitan 3^{10} celdas en la rejilla, de las que la mayor parte estarían vacías, para valores razonables de N .

La generalización de los árboles 2D a los árboles k D es también directa: simplemente se pasa en «ciclos» a través de las dimensiones (como se hizo para dos dimensiones alternando entre x e y) mientras se desciende por el árbol. Al igual que antes, en una situación aleatoria, los árboles resultantes tienen las mismas características que los árboles binarios de búsqueda. También como antes, hay una correspondencia natural entre estos árboles y el simple proceso geométrico. En tres dimensiones, la ramificación en cada nodo corresponde a cortar con un plano la región tridimensional de interés; en el caso general, se corta la región k -dimensional de interés con un hiperplano $(k - 1)$ -dimensional.

Si k es muy grande, probablemente haya fuertes desequilibrios en los árboles k D, una vez más porque los conjuntos naturales de puntos no pueden ser lo suficientemente densos como para mostrar aleatoriedad sobre un gran número de dimensiones. Por lo regular, todos los puntos de un subárbol tendrán los mismos valores para varias dimensiones, lo que conduce a muchas ramas de una sola vía. Una forma de limitar este problema es, en lugar de hacer ciclos sistemáticamente a través de las dimensiones, utilizar siempre la dimensión que mejor divida al conjunto de puntos. Esta técnica se puede aplicar también a los árboles 2D. Esto necesita que se almacene información extra (relativa a la dimensión de discriminación) en cada nodo, pero remedia el desequilibrio, en especial en árboles de dimensiones muy grandes.

En resumen, aunque es fácil ver la forma de generalizar los programas para

la búsqueda por rango de forma que puedan tratar problemas multidimensionales, no debe darse este paso a la ligera en aplicaciones muy grandes. Las grandes bases de datos con muchos atributos por registro pueden ser objetos verdaderamente complejos. A menudo es necesario tener una buena comprensión de las características de una determinada base de datos para poder desarrollar un método de búsqueda por rango que sea eficaz en una aplicación en particular. Éste es un problema de bastante importancia que todavía se está estudiando activamente.

Ejercicios

1. Escribir una versión no recursiva del programa `rango` de 1D dado en el texto.
2. Escribir un programa para listar todos los puntos de un árbol binario que *no* están en un intervalo dado.
3. Expresar el máximo y el mínimo número de celdas en las que podría buscarse en el método de la rejilla en función de las dimensiones de las celdas y del rectángulo de búsqueda.
4. Analizar la idea de evitar la búsqueda en celdas vacías utilizando listas enlazadas: cada celda podría estar enlazada con la siguiente celda no vacía de la misma fila y con la próxima celda no vacía de la misma columna. ¿De qué forma afectaría esta técnica al tamaño de la celda a utilizar?
5. Dibujar el árbol y la subdivisión del plano resultante de construir un árbol 2D para el ejemplo de puntos comenzando por una línea de división vertical.
6. Obtener un conjunto de puntos que conduzca al peor caso del árbol 2D que no tiene nodos con dos hijos; obtener la correspondiente subdivisión del plano.
7. Describir la forma de modificar cada uno de los métodos para que devuelva todos los puntos del interior de un círculo dado.
8. De todos los rectángulos de búsqueda de igual área, ¿qué figura es la que posiblemente haga que cada uno de los métodos se comporte peor?
9. ¿Qué método sería preferible para la búsqueda por rango cuando los puntos están agrupados en grandes conjuntos alejados entre sí?
10. Dibujar el árbol 3D que se obtiene al insertar los puntos (3,1,5), (4,8,3), (8,3,9), (6,2,7), (1,6,3), (1,3,5), (6,4,2) en un árbol inicialmente vacío.

Intersección geométrica

Un problema natural que aparece con frecuencia en aplicaciones que implican datos geométricos es el siguiente: «Dado un conjunto de N objetos, ¿son disjuntos dos a dos?» Los «objetos» en cuestión pueden ser segmentos, rectángulos, círculos, polígonos, o cualquier otro tipo de objetos geométricos. Cuando se trata de objetos físicos, se sabe que dos de ellos no pueden ocupar el mismo lugar al mismo tiempo, pero es bastante complejo escribir un programa de computadora que contemple este hecho. Por ejemplo, en un sistema para el diseño y realización de circuitos integrados o de tarjetas de circuitos impresos, es importante saber que dos cables no se cruzan para evitar un cortocircuito. En un sistema industrial para el diseño de plantillas por una máquina de corte por control numérico, es importante saber que no hay dos partes de la plantilla que se solapen. En el diseño gráfico por computadora, el problema de determinar qué partes de un conjunto de objetos están ocultas para un punto de vista particular, se puede formular como un problema de intersección geométrica de las proyecciones de los objetos sobre el plano de visión. Incluso en ausencia de objetos físicos, existen muchos ejemplos en los que la formulación matemática del problema conduce a un problema de intersección geométrica. En el Capítulo 43 se encontrará un ejemplo particularmente importante de esto.

La solución evidente al problema de la intersección consiste en comprobar si cada par de objetos se intersecan entre sí. Puesto que hay alrededor de $N^2/2$ pares de objetos, el tiempo de ejecución de este algoritmo es proporcional a N^2 . En algunas aplicaciones esto puede no ser un problema porque otros factores limitan el número de objetos a procesar. Sin embargo, en la mayor parte de los casos, es frecuente tener que considerar cientos de miles e incluso millones de objetos y el algoritmo de fuerza bruta N^2 es evidentemente inadecuado. En esta sección, se estudiará un método para determinar en un tiempo proporcional a $N \log N$ si dos objetos de un conjunto de N de ellos intersecan; este método se basa en los algoritmos presentados por M. Shamos y D. Hoey en un artículo fundamental de 1976.

En primer lugar se considerará un algoritmo para contar el número de in-

intersecciones de un conjunto de segmentos horizontales o verticales. Esto simplifica el problema en un sentido (los segmentos horizontales y verticales son objetos geométricos relativamente sencillos), pero lo complica en otro (contar todos los pares que se intersecan es más difícil que determinar la existencia de uno de ellos). Al igual que el capítulo anterior, se considerará el problema de contar (en lugar de estar obteniendo todas las respuestas) sólo para que el código sea menos engorroso —la generalización del método para obtener todos los pares que se intersecan es directa—. La implementación que se va a desarrollar combina los árboles de binarios de búsqueda y el programa de búsqueda por rango del capítulo anterior, en un programa doblemente recursivo.

Posteriormente se examinará el problema de determinar si dos segmentos de un conjunto de N de ellos se intersecan, sin restricciones en los mismos. Se puede aplicar la misma estrategia general que la utilizada para el caso horizontal-vertical. De hecho, la misma idea básica es válida para la detección de intersecciones entre muchos otros tipos de objetos geométricos. Sin embargo, para segmentos y otros objetos, la generalización para determinar todos los pares que se intersecan es algo más complicada que para el caso horizontal-vertical.

Segmentos horizontales y verticales

Para comenzar, se supondrá que todos los segmentos son horizontales o verticales: los dos puntos que definen cada segmento tienen igual coordenada x o igual coordenada y , como en los ejemplos de segmentos que se muestran en la Figura 27.1. (A esta restricción se la denomina a veces *geometría Manhattan* porque, al contrario de Broadway, el plano de las calles de Manhattan está formado casi exclusivamente por horizontales y verticales.) La restricción a los segmentos horizontales y verticales es ciertamente estricta, pero no por ello el problema se convierte en un «modelo reducido». Al contrario, esta restricción se impone a menudo en las aplicaciones particulares: por ejemplo, los circuitos integrados a gran escala se diseñan normalmente con esta restricción. En la figura de la derecha, los segmentos son relativamente cortos, como es normal en muchas aplicaciones, aunque por lo regular pueden encontrarse unos pocos segmentos muy largos.

El plan general del algoritmo para encontrar una intersección en tales conjuntos de segmentos consiste en imaginar el recorrido de una línea horizontal barriendo desde abajo hacia arriba. Las proyecciones sobre esta línea de barrido de los segmentos verticales son puntos y las de los segmentos horizontales son intervalos: a medida que la linea de barrido progresó hacia arriba, los puntos (que representan a los segmentos verticales) aparecen y desaparecen, y los segmentos horizontales aparecen periódicamente. Se encuentra una intersección cuando aparece un segmento horizontal, representado por un intervalo de la línea de barrido, que contiene a un punto que representa a un segmento vertical.

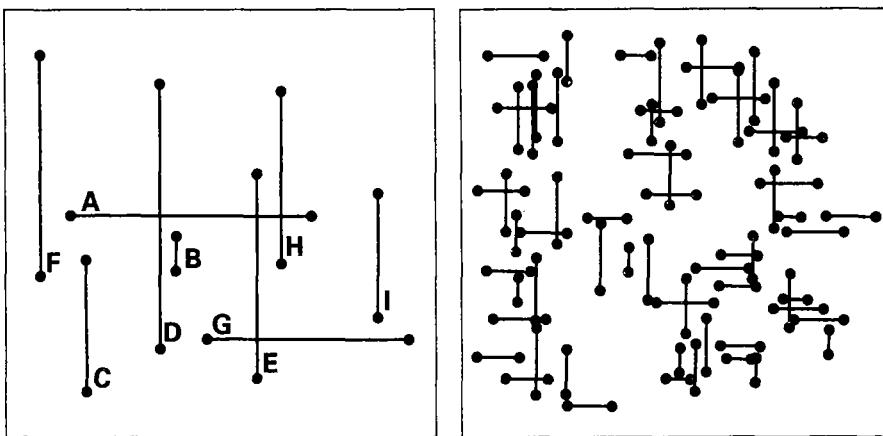


Figura 27.1 Problemas de intersección de dos segmentos (Manhattan).

Esta aparición significa que en este punto el segmento vertical interseca la línea de barrido, y que el segmento horizontal pertenece a esta línea, por lo que los dos segmentos, horizontal y vertical, deben cortarse. De esta forma, el problema bidimensional de encontrar un par de segmentos que se corten se reduce al unidimensional de búsqueda por rango del capítulo anterior.

Por supuesto, no es realmente necesario «barrer» todo el camino a lo largo del conjunto de segmentos; puesto que se necesita actuar sólo cuando se encuentren los puntos extremos de los segmentos, se puede comenzar ordenando los segmentos de acuerdo con su coordenada y , y procesar los segmentos en ese orden. Si se encuentra el punto del extremo inferior de un segmento vertical, se añade la coordenada x de ese segmento al árbol binario de búsqueda (denominado aquí el árbol x); si se encuentra el extremo superior de un segmento vertical, se suprime ese segmento (el x) del árbol; y si se encuentra un segmento horizontal, se hace una búsqueda de rango en el intervalo definido por sus dos coordenadas x . Como se verá, es preciso tener cierto cuidado al manejar coordenadas iguales en los puntos extremos de los segmentos (aunque el lector debería estar acostumbrado a encontrar dificultades como éstas en los algoritmos geométricos).

La Figura 27.2 muestra los primeros pasos del recorrido para encontrar las intersecciones del ejemplo de la izquierda de la Figura 27.1. El recorrido comienza en el punto con menor coordenada y , el extremo inferior de C. A continuación se encuentra E y luego D. El resto del proceso se muestra en la Figura 27.3; el próximo segmento que se encuentra es G, comprobándose si se interseca con C, D y E (los segmentos verticales que se intersecaron con la línea de barrido).

Para implementar el barrido, se necesita solamente ordenar los puntos ex-

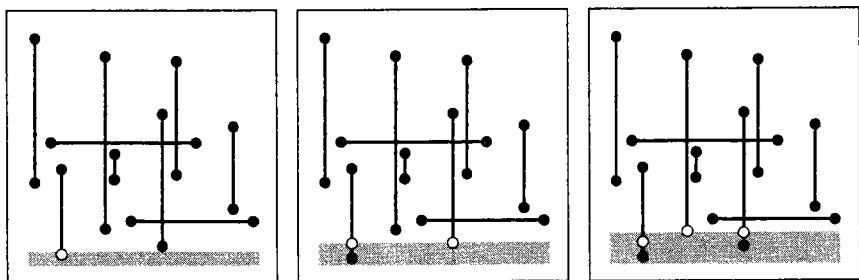


Figura 27.2 Búsqueda de intersecciones por barrido: pasos iniciales.

tremos de los segmentos por sus coordenadas y . Para el ejemplo, se obtiene la lista

C E D G I B F C H B A I E D H F

Cada segmento vertical aparece dos veces y cada segmento horizontal aparece una sola. Por las necesidades del algoritmo de intersección, esta lista ordenada se puede considerar como una serie de instrucciones de *insertar* (segmentos verticales cuando se encuentre el extremo inferior), *suprimir* (cuando se encuentre el extremo superior) y órdenes de *rango* (para los extremos de los segmentos horizontales). Todas estas «órdenes» son simplemente llamadas a las rutinas estándar de los árboles binarios de los Capítulos 14 y 26, utilizando las coordenadas x como claves.

La Figura 27.4 muestra el proceso de construcción del árbol x durante el barrido. Cada nodo del árbol corresponde a un segmento vertical —la clave utilizada para el árbol es la coordenada x —. Puesto que E está a la derecha, se encuentra en el subárbol derecho de C, etc. La primera línea de la Figura 27.4 corresponde a la Figura 27.2; el resto a la Figura 27.3.

Al encontrar un segmento horizontal, se utiliza para hacer una búsqueda por rango en el árbol: todos los segmentos verticales en el rango asociado a este segmento horizontal corresponden a intersecciones. En el ejemplo, se descubre la intersección entre E y G; después se insertan I, B y F. Luego se suprime C, se inserta H y se suprime B. Posteriormente se encuentra A, y se lleva a cabo la búsqueda por rango del intervalo definido por A, lo que descubre las intersecciones de A con D, E y H. A continuación se suprimen los extremos superiores de I, E, D, H y F, quedando el árbol vacío.

Implementación

El primer paso en la implementación es ordenar los puntos extremos de los segmentos por sus coordenadas y . Pero como se utilizan árboles binarios para

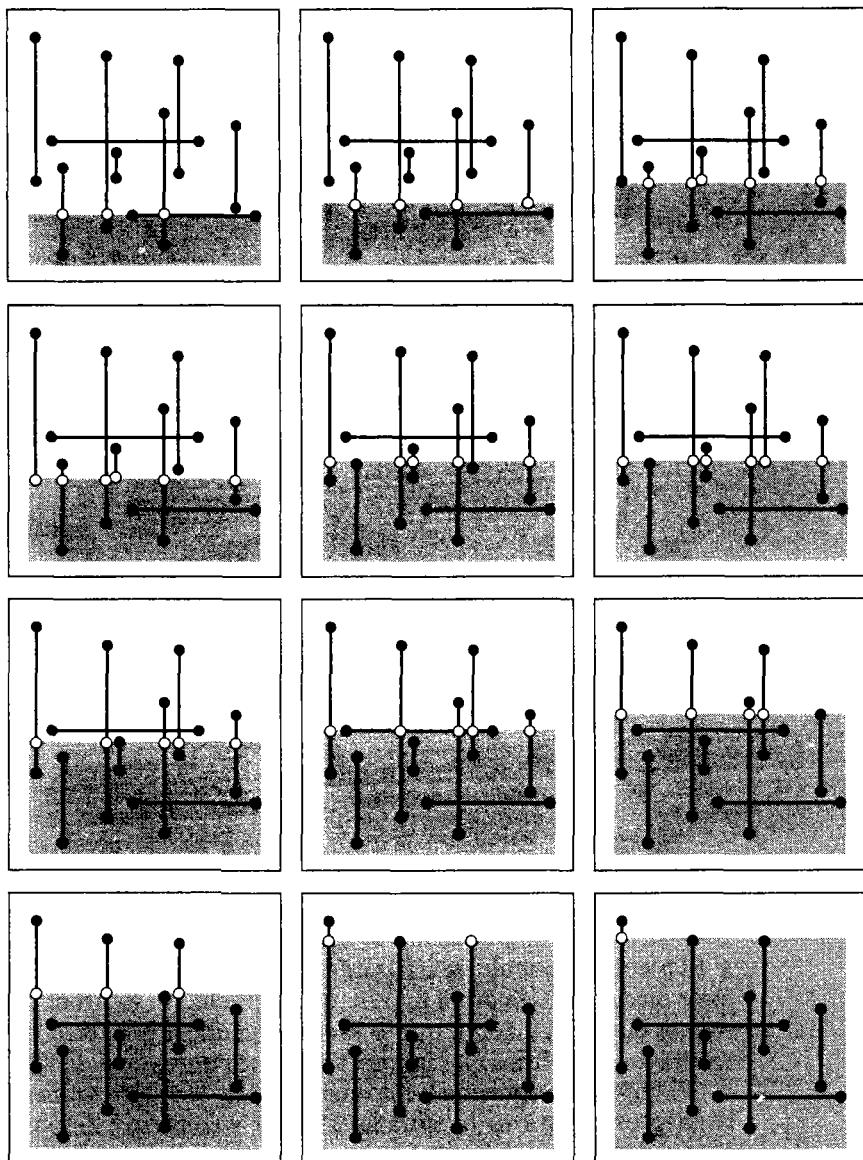


Figura 27.3 Búsqueda de intersecciones por barrido: final del proceso.

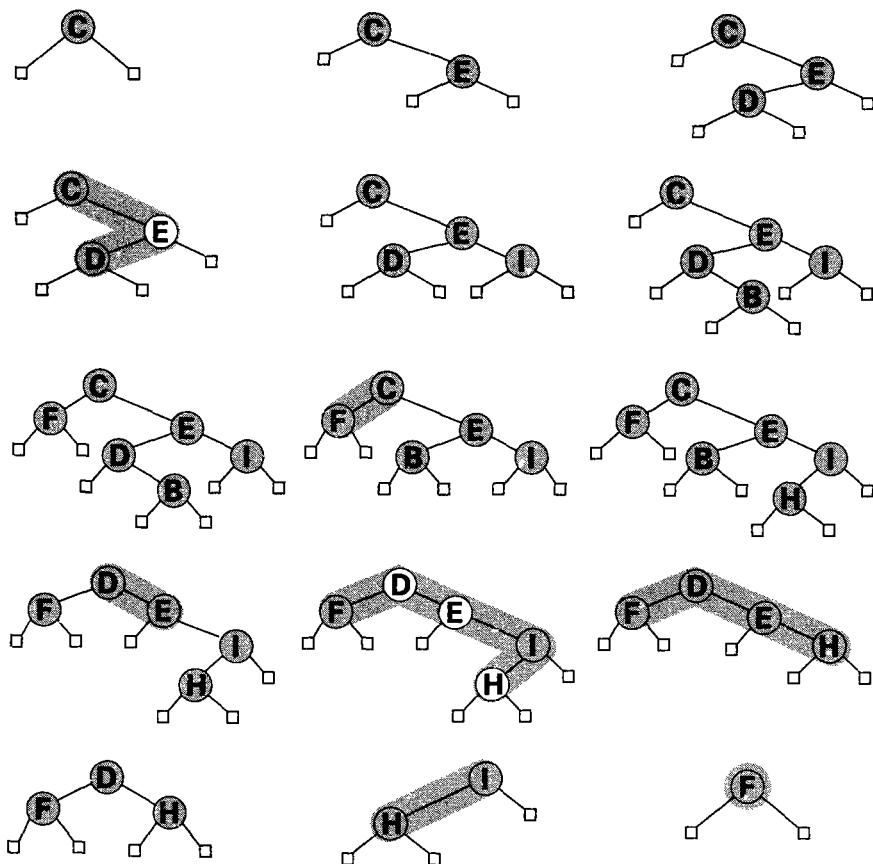


Figura 27.4 Estructura de datos durante el barrido: construcción del árbol x.

mantener la situación de los segmentos verticales con respecto a la línea horizontal de barrido, ¡se pueden utilizar también para la ordenación inicial de los y! Más concretamente, se utilizarán dos árboles binarios Xarbol e Yarbol de la clase diccionario de árbol binario del Capítulo 14. El árbol y contendrá los extremos de los segmentos, que se procesarán uno a uno, en orden; el árbol x contendrá los segmentos que intersecan la línea horizontal de barrido.

El programa siguiente lee primero grupos de cuatro números que definen los segmentos a partir de la entrada estándar y construye después el y árbol insertando las coordenadas y de los segmentos verticales y de los horizontales. Ahora el barrido en sí es efectivamente un recorrido en orden del Yarbol:

```
Dicc Xarbol(Nmax), Yarbol(Nmax);
struct segmento segmentos[Nmax];
int cuenta = 0;
```

```

int intersecciones()
{
    int x1, y1, x2, y2, N;
    for (N = 1; cin >> x1 >> y1 >>x2 >>y2; N++)
    {
        segmentos[N].p1.x = x1; segmentos[N].p1.y = y1;
        segmentos[N].p2.x = x2; segmentos[N].p2.y = y2;
        Yarbol.insertar(y1, N);
        if (y2 != y1) Yarbol.insertar(y2, N);
    }
    Yarbol.recorrer();
    return cuenta;
}

```

Para el conjunto de segmentos del ejemplo, se construye el árbol que se muestra en la Figura 27.5. El «ordenar según y » que necesita el algoritmo se efectúa con recorrer, que (Capítulo 14) llama al procedimiento visitar para cada uno de los nodos, en orden creciente de y . Todo el trabajo de encontrar las intersecciones (utilizando un árbol binario diferente sobre las coordenadas x) se realiza en el procedimiento visitar, que se especifica después.

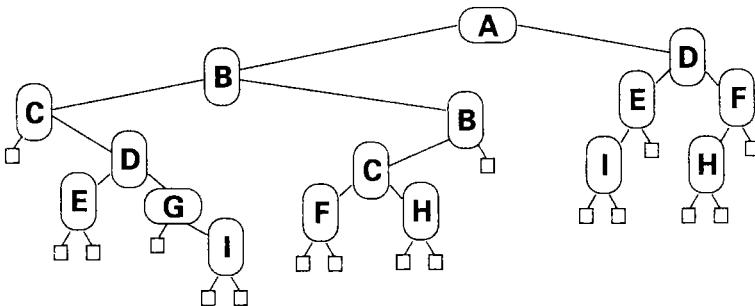


Figura 27.5 Ordenación para el barrido utilizando el árbol y .

A partir de la descripción del algoritmo, es fácil poner el código en el punto donde se «visita» cada nodo:

```

void Dicc::visitar(tipoElemento v, tipoInfo info)
{
    int t, x1, x2, y1, y2;
    x1 = segmentos[info].p1.x; y1 = segmentos[info].p1.y;
    x2 = segmentos[info].p2.x; y2 = segmentos[info].p2.y;
    if (x2 < x1) { t = x2; x2 = x1; x1 = t; }
    if (y2 < y1) { t = y2; y2 = y1; y1 = t; }

```

```

if (v == y1)
    Xarbol.insertar(x1, info);
if (v == y2)
{
    Xarbol.suprimir(x1, info);
    cuenta += Xarbol.rango(x1, x2);
}
}

```

En primer lugar, se extraen las coordenadas de los extremos del segmento correspondiente del array `segmentos`, indexado por el campo `info` del nodo. Luego se compara el campo `clave` del nodo con estas coordenadas para determinar cuándo este nodo corresponde a una extremidad superior o inferior del segmento: si es el extremo inferior, se inserta en el árbol x , y, si es el superior, se suprime del árbol x y se lleva a cabo una búsqueda por rango. La implementación anterior difiere ligeramente de esta descripción en que los segmentos horizontales se insertan realmente en el árbol x , y se suprimen luego inmediatamente, y se efectúa, para los segmentos verticales, una búsqueda por rango en un intervalo reducido a un punto. Esto hace que el código trate adecuadamente el caso de segmentos verticales que se solapan, que se consideran que se van a «intersecar».

Esta aproximación de la aplicación combinada de procedimientos recursivos que opera sobre las coordenadas x e y es muy importante en los algoritmos geométricos. Otro ejemplo de ella es el algoritmo de árbol 2D del capítulo anterior, y además se verá otro en el próximo capítulo.

Propiedad 27.1 *Todas las intersecciones entre N segmentos horizontales y verticales se pueden encontrar en un tiempo proporcional a $N \log N + I$, siendo I el número de intersecciones.*

Las operaciones de manipulación de árbol tardan un tiempo proporcional a $\log N$, por término medio (si se utilizan árboles equilibrados, podría garantizarse un peor caso en $\log N$), pero el tiempo que se emplea en la búsqueda por rango también depende del número total de intersecciones. En general, este número puede ser muy grande. Por ejemplo, si se tienen $N/2$ segmentos horizontales y $N/2$ segmentos verticales distribuidos en un modelo entrecruzado, entonces el número de intersecciones es proporcional a N^2 .■

Como en la búsqueda por rango, si se conoce por adelantado que el número de intersecciones es muy grande, debería utilizarse alguna variante de fuerza bruta. Por lo regular, las aplicaciones presentan situaciones del tipo «buscar una aguja en un pajar», donde se debe examinar un gran conjunto de segmentos para no encontrar más que unas pocas intersecciones.

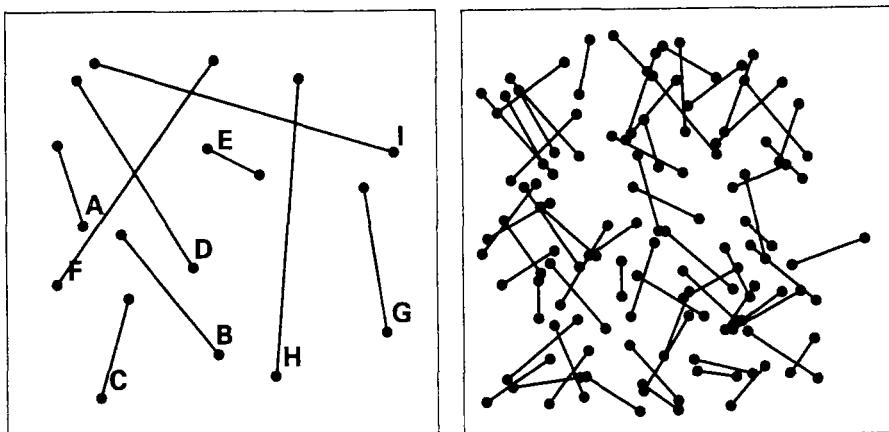


Figura 27.6 Problemas de intersección de dos segmentos cualesquiera.

Intersección de segmentos en general

Cuando se permiten segmentos de inclinación arbitraria, la situación se vuelve más complicada, como se ilustra en la Figura 27.6. Primero, las distintas orientaciones de los segmentos posibles hacen necesario preguntar explícitamente cuándo se intersecan ciertos pares de segmentos, lo que no se puede resolver con una simple verificación de búsqueda por rango. Segundo, la relación de orden entre segmentos para el árbol binario es más complicada que antes, puesto que depende del intervalo actual de y . Tercero, cualquier intersección que se produzca añadirá nuevos valores «interesantes» de y , que posiblemente serán diferentes del conjunto de valores de y correspondientes a los extremos de los segmentos.

Resulta que estos problemas se pueden manejar en un algoritmo con la misma estructura básica que la dada anteriormente. Para simplificar la presentación, se considerará un algoritmo para detectar cuándo existe o no un par de segmentos que se intersecan en un conjunto de N segmentos, y posteriormente se presentará cómo generalizarlo para detectar todas las intersecciones.

Como antes, primero se ordena sobre y para dividir el espacio en franjas dentro de las que no aparece ningún punto extremo de segmento. Exactamente como antes, se procede a lo largo de la lista ordenada de puntos, añadiendo cada segmento a un árbol binario de búsqueda cuando se encuentre su extremo inferior y suprimiéndolo cuando se encuentre su extremo superior. También como antes, el árbol binario da el orden en el que aparecen los segmentos en la «franja» horizontal entre dos valores y consecutivos. Por ejemplo, en la franja entre el extremo inferior de D y el superior de B de la Figura 27.6, los segmentos deben aparecer en el orden F B D H G. Se supone que no hay intersecciones dentro

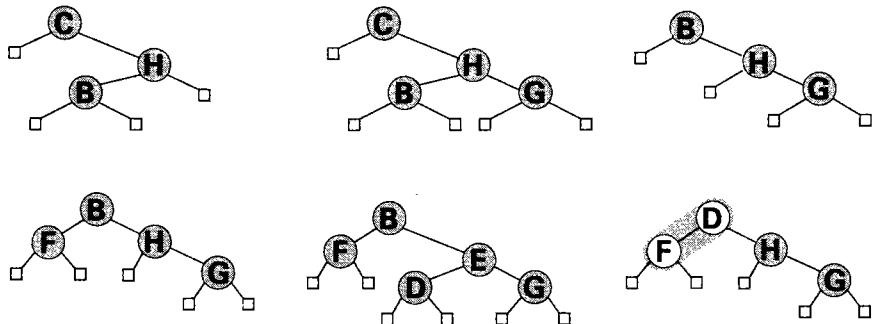


Figura 27.7 Estructura de datos (árbol x) para el problema general.

de la franja horizontal actual: el objetivo es mantener esta estructura de árbol y utilizarla para encontrar la primera intersección.

Para construir el árbol, no se puede utilizar solamente como claves a las coordenadas x de los extremos de los segmentos (por ejemplo, si se hace esto en el ejemplo anterior se invertiría el orden real de B y D). En su lugar se utilizará una relación de orden más general: se dice que un segmento x está a la derecha de un segmento y si los dos extremos de x están del mismo lado de y que un punto del infinito situado a la derecha, *o bien* si y está a la izquierda de x , definiendo «izquierda» de forma similar. Así, en el diagrama anterior, B está a la derecha de A y B está a la derecha de C (puesto que C está a la izquierda de B). Si x no está ni a la derecha ni a la izquierda de y , entonces los dos segmentos deben intersecarse. Esta operación generalizada de «comparación de segmentos» se puede implementar utilizando el procedimiento `ccw` del Capítulo 24. Excepto al utilizar esta función siempre que se necesite una comparación, se pueden utilizar sin modificación los procedimientos estándar de árbol binario de búsqueda (incluso si se desea árboles equilibrados). La Figura 27.7 muestra la evolución del árbol del ejemplo entre el momento en el que se encontró el segmento C y en el que se encontró el D. Cada «comparación» que se lleva a cabo durante los procedimientos de manipulación de árboles es realmente una comprobación de intersección de segmentos: si el procedimiento de búsqueda en árbol binario no puede decidir si hay que ir a la derecha o a la izquierda, los dos segmentos en curso se deben intersecar, y ya está todo hecho.

Pero ésta no es la historia completa, porque esta operación de comparación generalizada no es *transitiva*. En el ejemplo anterior, F está a la izquierda de B (porque B está a la derecha de F) y B está a la izquierda de D, pero F *no* está a la izquierda de D. Es esencial destacar este problema, porque el procedimiento de suprimir en el árbol binario supone que la operación de comparación es transitiva: cuando se suprime B del último árbol de la serie anterior, se obtiene el árbol de la Figura 27.7 sin ninguna comparación explícita de F y D. Para que el algoritmo de comprobación de intersección sea correcto, se debe verificar ex-

plícitamente que las comparaciones son válidas cada vez que se cambia la estructura del árbol. En concreto, cada vez que se hace que el enlace izquierdo del nodo x apunte al nodo y , se comprueba explícitamente si el segmento correspondiente a x está a la izquierda del segmento correspondiente a y , de acuerdo con la definición anterior, y lo mismo para la derecha. Por supuesto, esta comparación podría entrañar la detección de una intersección, como es el caso del ejemplo.

En resumen, para detectar una intersección en un conjunto de N segmentos, se utiliza el programa anterior, pero se suprime la llamada a `rango` y se generalizan las rutinas de árbol binario para que permitan comparaciones generalizadas como las descritas anteriormente. Si no hay intersección, se comienza con un árbol vacío y se finaliza con el mismo árbol vacío sin encontrar segmentos no comparables. Si hay una intersección, entonces se deben comparar entre sí los dos segmentos que se intersecan en algún momento del proceso de barrido y se descubrirá la intersección.

Sin embargo, una vez que se ha encontrado una intersección no se debe simplemente continuar y esperar a encontrar al resto, porque los dos segmentos que se intersecan deben intercambiar su lugar en el orden, inmediatamente después del punto de intersección. Una forma de realizar esta operación sería utilizar una cola de prioridad en lugar de un árbol binario para la ordenación de y : inicialmente se ponen los segmentos en la cola de prioridad de acuerdo con las coordenadas y de sus extremos, y luego se efectúa un barrido ascendente tomando sucesivamente la coordenada y más pequeña de la cola de prioridad y haciendo una inserción o supresión en el árbol binario, como se hizo antes. Cuando se encuentra una intersección, se añaden nuevas entradas en la cola de prioridad, una por cada segmento, utilizando para cada una el punto de intersección como extremo inferior.

Otra forma de encontrar todas las intersecciones, que es apropiada si no se espera que haya muchas, es simplemente eliminar uno de los segmentos cuando se detecta una intersección. Una vez efectuado el barrido, se sabe que todos los pares que se intersequen deben englobar a uno de esos segmentos, y se puede utilizar un método de fuerza bruta para enumerar todas las intersecciones.

Propiedad 27.2 *Todas las intersecciones entre N segmentos se pueden encontrar en un tiempo proporcional a $(N+I)\log N$, donde I es el número de intersecciones.*

Esto es consecuencia directa de la descripción anterior.■

Una característica interesante del procedimiento anterior es que se puede adaptar, cambiando el procedimiento general de comparación, para detectar la existencia de un par de intersecciones en un conjunto de figuras geométricas más generales. Por ejemplo, si se implementa un procedimiento para comparar dos rectángulos cuyos lados son paralelos horizontal y verticalmente, de acuerdo con la regla trivial de que un rectángulo x está a la izquierda de un rectángulo y si

el lado derecho de x está a la izquierda del lado izquierdo de y , entonces se puede utilizar el método anterior para detectar las intersecciones en un conjunto de rectángulos de este tipo. Para círculos, se puede utilizar las coordenadas x de los centros para la ordenación y efectuar explícitamente comprobaciones de intersección (por ejemplo, comparar la distancia entre los centros con la suma de los radios). Una vez más, si esta comparación se utiliza en el método anterior, se tiene un algoritmo de comprobación de intersecciones en un conjunto de círculos. El problema de detectar todas las intersecciones en estos casos es mucho más complicado, aunque el método de fuerza bruta ya mencionado en el párrafo anterior es válido siempre que se esperen pocas intersecciones. Otra aproximación que es suficiente en muchas aplicaciones consiste en considerar los objetos complejos como conjuntos de segmentos y utilizar el procedimiento de intersección de segmentos.

Ejercicios

1. ¿Cómo se determinaría si se intersecan dos triángulos? ¿Y dos cuadrados? ¿Y dos polígonos regulares de n lados, con $n > 4$?
2. En el algoritmo de intersección de segmentos horizontales y verticales, ¿cuántos pares de segmentos se deben comprobar en un conjunto de segmentos sin intersección, en el peor caso? Mostrar un diagrama que apoye la respuesta.
3. ¿Qué sucede cuando se utiliza el procedimiento de intersección de segmentos horizontales y verticales en un conjunto de segmentos con inclinaciones arbitrarias?
4. Escribir un programa para encontrar el número de pares de intersecciones en un conjunto de N segmentos aleatorios horizontales y verticales, si cada segmento se genera por dos coordenadas enteras aleatorias entre 0 y 1.000 y un bit aleatorio para distinguir si es vertical u horizontal.
5. Dar un método para comprobar si un polígono es simple (no se interseca consigo mismo).
6. Dar un método para averiguar si un polígono está contenido totalmente dentro de otro.
7. Describir cómo se resolvería el problema general de intersección de segmentos dado el hecho adicional de que la separación mínima entre dos segmentos es mayor que la longitud máxima de los segmentos.
8. Obtener las estructuras de árbol binario que existen cuando el algoritmo de intersección de segmentos detecta la intersección en los segmentos de la Figura 27.6, si se ha hecho una rotación de 90 grados.
9. ¿Son transitivos los procedimientos de comparación de círculos y rectángulos Manhattan descritos en el texto?
10. Escribir un programa para encontrar el número de pares que se intersecan en un conjunto de N segmentos aleatorios, si cada segmento está generado por coordenadas enteras aleatorias entre 0 y 1.000.

Problemas del punto más cercano

Por lo regular, en los problemas geométricos relativos a puntos del plano interviene el cálculo implícito o explícito de las distancias entre los puntos. Por ejemplo, un problema muy natural que se presenta en muchas aplicaciones es el del *vecino más próximo*: encontrar, entre los puntos de un conjunto dado, el más cercano a un nuevo punto también dado. Parece necesario comparar las distancias entre el punto dado y cada punto del conjunto, pero existen soluciones mucho mejores. En esta sección se verán otros problemas de distancia, un prototipo de algoritmo y una estructura geométrica fundamental denominada *diagrama de Voronoi*, que se puede utilizar con efectividad en una gran variedad de problemas de este tipo en el plano. Se hará una aproximación que consistirá en describir un método general de resolución de problemas del punto más cercano por medio de un análisis cuidadoso del prototipo de implementación de un problema sencillo.

Algunos de los problemas que se considerarán en este capítulo son similares a los de búsqueda por rango del Capítulo 26, y los métodos de la rejilla y de los árboles 2D ya estudiados son también adecuados para resolver el problema del vecino más próximo o de otros varios. Sin embargo, el defecto fundamental de tales métodos es que se apoyan en la aleatoriedad del conjunto de puntos: tienen un mal rendimiento en el peor caso. El objetivo de este capítulo es examinar otra aproximación general que garantice un buen rendimiento para muchos problemas, sin importar cuál sea la entrada. Algunos de los métodos son demasiado complicados para que se pueda examinar aquí la implementación completa, e implican un sobrecoste tan grande que incluso los métodos más simples pueden ser más eficaces cuando el conjunto de puntos no es muy grande o está bastante bien distribuido. Sin embargo, el estudio de los métodos que presentan un buen rendimiento en el peor caso revelará algunas de las propiedades fundamentales de los conjuntos de puntos que deben ser comprendidas,

incluso aunque los métodos más simples parezcan más convenientes en algunas situaciones específicas.

La aproximación general que se examinará proporciona otro ejemplo de la utilización de procedimientos doblemente recursivos para entrelazar el procesamiento entre las dos direcciones de coordenadas. Los dos métodos de este tipo que se han visto anteriormente (árboles kD e intersección de segmentos) se fundan en árboles binarios de búsqueda; aquí el método es del tipo «combina y vencerás» basado en la ordenación por fusión.

Problema del par más cercano

El problema del *par más cercano* consiste en encontrar los dos puntos más cercanos entre sí de un conjunto de puntos dado. Este problema está relacionado con el del vecino más próximo; aunque no sea de aplicación general, servirá como prototipo de los problemas del punto más cercano en el sentido de que se puede resolver con un algoritmo cuya estructura recursiva general se adapte a otros problemas de este tipo.

Para encontrar la distancia mínima entre dos puntos, parecería necesario examinar las distancias entre todos los pares de puntos: para N puntos esto podría significar un tiempo de ejecución proporcional a N^2 . Sin embargo, es posible utilizar una ordenación para examinar sólo alrededor de $N \log N$ distancias entre puntos en el peor caso (algunos menos por término medio) y obtener un peor caso proporcional a $N \log N$ (mucho menos en el caso medio). En esta sección se examinará con detalle un algoritmo como éste.

El algoritmo que se utilizará está basado en una estrategia directa de «divide y vencerás». La idea es ordenar los puntos según una coordenada, por ejemplo la x , y utilizar después esa ordenación para dividir los puntos en dos mitades. El par más cercano del conjunto completo es o bien el de una de las dos mitades o el formado por un elemento de cada mitad. El caso interesante, por supuesto, es cuando el par más cercano cruza la línea divisoria: el par más cercano de cada mitad se puede encontrar fácilmente utilizando llamadas recursivas, pero ¿cómo se pueden comprobar eficazmente todos los pares cuyos elementos que están uno a cada lado de la línea divisoria?

Puesto que la única información que se busca es el par más cercano al conjunto de puntos, solamente se necesita examinar los puntos que están dentro de la distancia \min de la línea divisoria, siendo \min la menor de las distancias entre los pares más cercanos encontrados en las dos mitades. Sin embargo, esta observación no es por sí misma ayuda suficiente en el peor caso, puesto que puede haber muchos pares de puntos muy cercanos a la línea divisoria; por ejemplo, todos los puntos de cada mitad podrían estar situados en la proximidad de la línea divisoria.

Para tratar tales situaciones, parece necesario ordenar los puntos según y . Después se puede limitar el número de cálculos de distancias que implican a

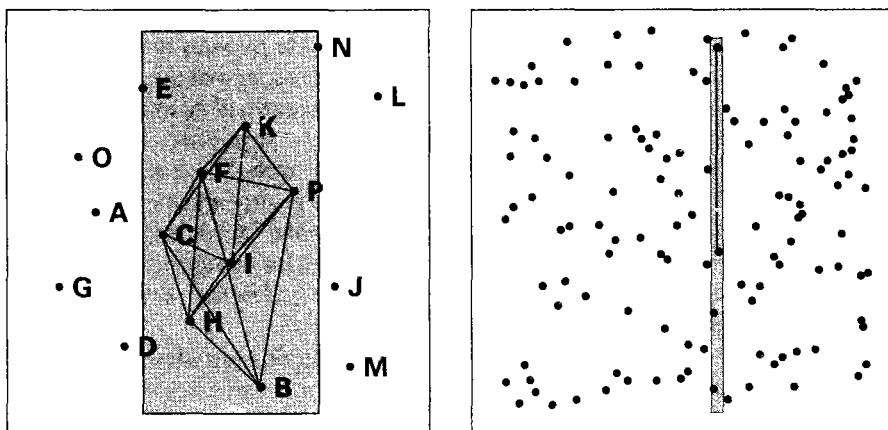


Figura 28.1 Aproximación de divide y vencerás para encontrar el par más cercano.

cada punto de la siguiente forma: recorriendo los puntos en orden creciente de y , se comprueba si cada uno está dentro de la franja vertical que contiene a todos los puntos del plano situados a menos de la distancia \min de la línea divisoria. Para cada punto que pase la comprobación, se calcula la distancia entre él y otro cualquiera situado también dentro de la franja cuya coordenada y sea menor que la y del punto en cuestión, pero en una cantidad que no sea mayor que \min . El hecho de que la distancia entre todos los pares de puntos de cada mitad sea inferior a \min significa que posiblemente se comprueben sólo unos pocos puntos.

En el pequeño conjunto de puntos de la izquierda de la Figura 28.1, la línea vertical divisoria imaginaria inmediatamente a la derecha de F tiene ocho puntos a la izquierda y ocho a la derecha. El par más cercano de la mitad izquierda es AC (o AO) y el de la derecha es JM. Si se ordenan los puntos según y , entonces el par más cercano dividido por la línea se encuentra comparando los pares HI, CI, FK (el par más cercano del conjunto completo de puntos), y finalmente EK. Para conjuntos de puntos más grandes, la banda que podría contener un par más cercano sobre la línea divisoria es más estrecha, como se muestra a la derecha de la Figura 28.1.

Aunque este algoritmo es simple, se debe tener cierto cuidado para implementarlo eficazmente: por ejemplo, sería muy costoso ordenar los puntos según y en el interior de la rutina recursiva. Se han visto varios algoritmos con tiempos de ejecución descritos por la recurrencia $C_N = 2C_{N/2} + N$, lo que implica que C_N es proporcional a $N\log N$; si se hubiera hecho la ordenación completa sobre y , entonces la recurrencia devendría $C_N = 2C_{N/2} + N\log N$, lo que implica que C_N es proporcional a $N\log^2 N$ (ver el Capítulo 6). Para eludir esto se necesita evitar la ordenación según y .

La solución a este problema es simple, pero sutil. El método ordenfusion

del Capítulo 12 se basa en dividir los elementos a ordenar exactamente como se dividieron los puntos anteriormente. Hay dos problemas a resolver y el mismo método general de resolución, ¡así que se pueden resolver al mismo tiempo! Más concretamente, se escribirá una rutina recursiva que ordene según y y encuentre el par más cercano. Esto lo hará dividiendo al conjunto de puntos por la mitad, llamándose a sí misma recursivamente para ordenar las dos mitades según y y encontrar el par más cercano de cada mitad, fusionando después para completar la ordenación sobre y y aplicando el procedimiento anterior para completar el cálculo del par más cercano. De esta forma, se evita el coste de hacer una ordenación extra de y al entremezclar los movimientos de datos que se requieren para la ordenación con los que se requieren para el cálculo del par más cercano.

Para la ordenación y , la división en dos mitades se podría hacer de cualquier manera, pero, para el cálculo del par más cercano, se necesita que los puntos de una mitad tengan todas coordenadas x más pequeñas que los puntos de la otra. Esto se lleva a cabo fácilmente ordenando según x antes de hacer la división. De hecho, ¡se puede utilizar la misma rutina para ordenar según x ! Una vez que se acepta este plan general, la implementación no es difícil de entender.

Como se mencionó anteriormente, la implementación utilizará los procedimientos recursivos ordenar y fusión del Capítulo 12. El primer paso consiste en modificar las estructuras de lista para que contengan puntos en lugar de claves, y modificar fusión de forma que compruebe una variable global pasada para decidir qué tipo de comparación hacer. Si pasada vale 1 se deberían comparar las coordenadas x de los dos puntos; si pasada vale 2 se comparan las coordenadas y . La implementación es directa:

```
int comp(struct nodo *t)
{ return (pasada == 1) ? t->p.x : t->p.y; }
struct nodo *fusion(struct nodo *a, struct nodo *b)
{
    struct nodo *c;
    c = z;
    do
        if (comp(a) < comp(b))
            { c->siguiente = a; c = a; a = a->siguiente; }
        else
            { c->siguiente = b; c = b; b = b->siguiente; }
    while (c != z);
    c = z->siguiente; z->siguiente = z;
    return c;
}
```

El nodo ficticio z que aparece al final de cada lista se inicializa para contener un punto «centinela» con coordenadas x y y artificialmente grandes.

Para evaluar las distancias, se utiliza otro procedimiento simple que verifica si la distancia entre los dos puntos pasados como argumentos es inferior a la variable global `min`. Si lo es, se asigna esta distancia a `min` y se guardan los puntos en las variables globales `pc1` y `pc2`:

```
comprobar(struct punto p1, struct punto p2)
{
    float dist;
    if ((p1.y != z->p.y) && (p2.y != z->p.y))
    {
        dist = sqrt((p1.x-p2.x)*(p1.x-p2.x) +
                    (p1.y-p2.y)*(p1.y-p2.y));
        if (dist < min)
            { min = dist; pc1 = p1; pc2 = p2; };
    }
}
```

Así, la variable global `min` contiene siempre la distancia entre `pc1` y `pc2`, el par más cercano encontrado hasta ahora.

El siguiente paso es modificar la función recursiva `ordenar` del Capítulo 12 para hacer también el cálculo del punto más cercano cuando pasada es 2, de la siguiente forma:

```
struct nodo *ordenar(struct nodo *c, int N)
{
    int i;
    struct nodo *a, *b;
    float medio;
    struct punto p1, p2, p3, p4;
    if (c->siguiente == z) return c;
    a = c;
    for (i = 2; i <= N/2; i++) c = c->siguiente;
    b = c->siguiente; c->siguiente = z;
    if (pasada == 2) medio = b->p.x;
    c = fusion(ordenar(a, N/2), ordenar(b, N-(N/2)));
    if (pasada == 2)
    {
        p1 = z->p; p2 = z->p; p3 = z->p; p4 = z->p;
        for (a = c; a != z; a = a->siguiente)
            if (fabs(a->p.x - medio) < min)
            {
                comprobar(a->p, p1);
            }
    }
}
```

```

        comprobar(a->p, p2);
        comprobar(a->p, p3);
        comprobar(a->p, p4);
        p1 = p2; p2 = p3; p3 = p4; p4 = a->p;
    }
}
return c;
}

```

Si pasada vale 1, ésta es exactamente la rutina recursiva de ordenación por fusión del Capítulo 12: devuelve una lista enlazada que contiene los puntos ordenados por sus coordenadas x (porque fusion ha sido modificado como se describió antes para comparar las coordenadas x en la primera pasada). La magia de esta implementación se manifiesta cuando pasada vale 2. El programa no sólo ordena según y (porque fusion ha sido modificado como se describió anteriormente para comparar las coordenadas y en la segunda pasada), sino que también efectúa el cálculo del punto más cercano. *Antes* de las llamadas recursivas, los puntos están ordenados según x : esta ordenación se utiliza para dividir los puntos en dos mitades y encontrar la coordenada x de la línea divisoria. *Después* de las llamadas recursivas se ordenan los puntos según y , y se sabe que la distancia entre todo par de puntos de cada mitad es mayor que \min . La ordenación sobre y se utiliza para recorrer los puntos cercanos a la línea divisoria; el valor de \min se utiliza para limitar el número de puntos que se deben comprobar. Cada punto situado a menos de una distancia \min de la línea divisoria se compara con cada uno de los cuatro puntos encontrados previamente dentro de una distancia \min de la linea divisoria. Esta comparación garantiza encontrar todos los pares de puntos que están a una distancia inferior a \min a cada lado de la línea divisoria.

¿Por qué se comparan los *cuatro* últimos puntos y no los dos, tres o cinco? Esto es una particularidad geométrica sorprendente que quizás el lector desee comprobar: se sabe que los puntos situados en un mismo lado de la línea divisoria están separados por al menos \min , por lo que el número de puntos incluidos en cualquier círculo de radio \min es limitado. Se podrían comprobar más de cuatro puntos, pero no es difícil convencerse de que cuatro es suficiente.

El código siguiente llama a ordenar dos veces para efectuar el cálculo del par más cercano. Primero, se ordena según x (con pasada igual a 1); después se ordena según y , y se encuentra el par más cercano (con pasada igual a 2):

```

z = new nodo;
z->p.x = max; z->p.y = max; z->siguiente = z;
h = new nodo; h->siguiente = leerlista();
min = max;
pasada = 1; h->siguiente = ordenar(h->siguiente, N);
pasada = 2; h->siguiente = ordenar(h->siguiente, N);

```

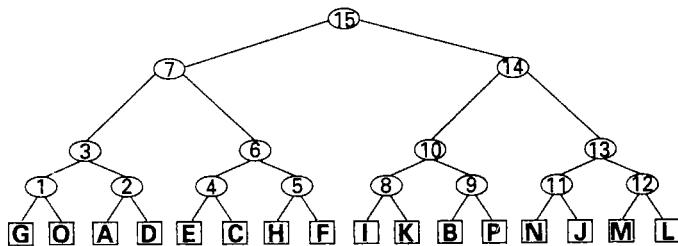


Figura 28.2 Árbol de llamadas recursivas para el cálculo del par más cercano.

Después de estas llamadas, el par de puntos más cercanos se encuentra en las variables globales `pc1` y `pc2`, que controla el procedimiento `comprobar «de encontrar el mínimo»`.

La Figura 28.2 muestra el árbol de llamadas recursivas que describe el funcionamiento de este algoritmo en el pequeño ejemplo del conjunto de puntos. Un nodo interno de este árbol representa una línea vertical que divide a los puntos del subárbol izquierdo y del derecho. Los nodos están numerados en el orden en el que se examinan las líneas verticales en el algoritmo. Esta numeración corresponde a un recorrido en orden posterior del árbol porque el cálculo que implica a la línea divisoria tiene lugar *después* de las llamadas recursivas, y es simplemente otra forma de ver el orden en el que se hace la fusión durante una ordenación por fusión recursiva (ver el Capítulo 12).

De este modo, primero se trata la línea entre G y O y se retiene el par GO como el más cercano, por ahora. Luego se trata la línea entre A y D, pero A y D están demasiado alejados como para modificar el valor de `min`. A continuación se trata la línea entre O y A y GD; GA y OA son los pares más cercanos sucesivos. En este ejemplo se comprueba que no se encuentran pares más cercanos hasta FK, que es el último par comprobado en la última línea divisoria tratada.

El lector que siga cuidadosamente el desarrollo puede notar que no se ha implementado el algoritmo puro de divide y vencerás descrito con anterioridad —en realidad no se calcula el par más cercano de cada una de las dos mitades, tomando luego el mejor de los dos—. En lugar de esto, se obtiene el más cercano de los dos pares más próximos utilizando simplemente la variable global `min` durante el cálculo recursivo. Cada vez que se encuentra un par más cercano, se considera de hecho una franja vertical más estrecha alrededor de la línea divisoria actual, sin tener en cuenta en qué punto se encuentra el cálculo recursivo.

La Figura 28.3 muestra este proceso detalladamente. La coordenada *x* de estos diagramas se ha ampliado para resaltar la orientación *x* del proceso y poner en evidencia el paralelismo con la ordenación por fusión (ver Capítulo 12). Se comienza haciendo una ordenación *y* sobre los cuatro puntos más a la izquierda, G O A D, ordenando G O, luego A D y fusionando después los resul-

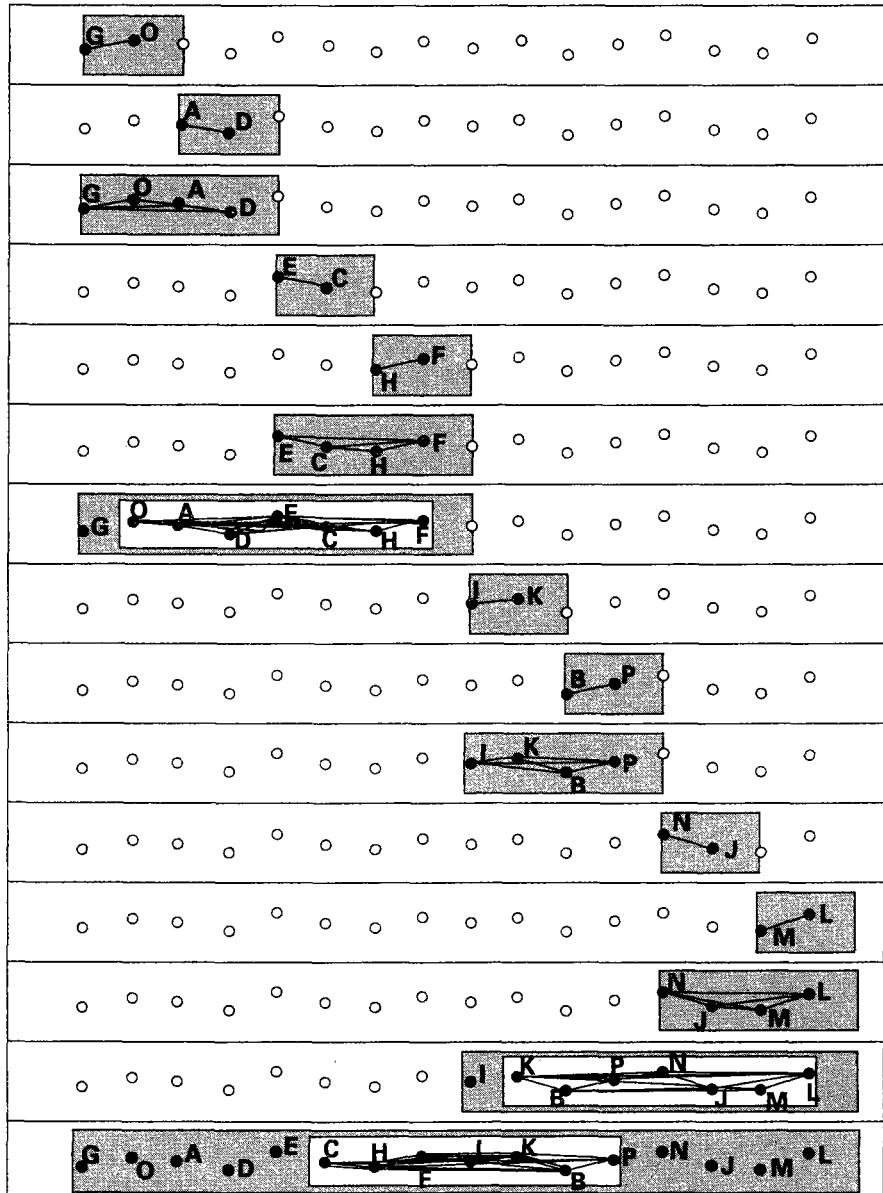


Figura 28.3 Cálculo del par más cercano (coordenada x ampliada).

tados. Después de la fusión, se completa la ordenación y encontrándose el par más cercano AO. A continuación se hace lo mismo con E C H F, etcétera.

Propiedad 28.1 *El par más cercano de un conjunto de N puntos se puede encontrar con un número de pasos en O(NlogN).*

Esencialmente, el cálculo se realiza en el tiempo de hacer dos ordenaciones por fusión (una sobre las coordenadas x , y otra sobre las y) más el coste del recorrido a lo largo de la línea divisoria. Este coste está también gobernado por la recurrencia $T_N = 2T_{N/2} + N$.■

La estrategia general que se ha utilizado aquí para el problema del par más cercano puede servir para resolver otros problemas geométricos. Por ejemplo, otro caso de interés es el de *todos los vecinos más próximos*: para cada punto se desea encontrar el punto más próximo a él. Este problema se puede resolver utilizando un programa como el anterior y añadiendo un procesamiento extra a lo largo de la línea divisoria para determinar, para cada punto, si existe un punto homólogo situado en la otra mitad, más cercano que el más cercano de los de su propia mitad. De nuevo la «libre» ordenación en y es útil para este cálculo.

Diagramas de Voronoi

El conjunto de todos los puntos más cercanos a un punto dado que todos los otros puntos en un conjunto de puntos es una interesante estructura geométrica denominada *polígono de Voronoi* del punto. La unión de todos los polígonos de Voronoi de un conjunto de puntos se denomina *diagrama de Voronoi*. Esto es lo máximo en el cálculo del punto más cercano: se verá que la mayoría de los problemas tratados que implican distancias entre puntos admiten soluciones naturales e interesantes basadas en los diagramas de Voronoi. Los diagramas para el ejemplo del conjunto de puntos se muestran en la Figura 28.4.

El polígono de Voronoi de un punto está formado por las mediatrices de los segmentos que enlazan al punto con los que le son más cercanos. Su definición real se hace de otra forma: el polígono de Voronoi se define como el perímetro del conjunto de todos los puntos del plano más cercanos al punto dado que a cualquier otro punto del conjunto de puntos, y cada lado del polígono de Voronoi separa al punto en cuestión de cada uno de los puntos más «cercanos a él».

El *dual* del diagrama de Voronoi, que se muestra en la Figura 28.5, hace explícita esta correspondencia: en el dual, se dibuja un segmento entre cada punto y todos los puntos «cercanos» a él. Esta estructura se denomina también triangulación de Delaunay. Los puntos x y y se enlazan en el dual de Voronoi solamente si sus polígonos de Voronoi tienen un lado en común.

El diagrama de Voronoi y la triangulación de Delaunay tienen muchas pro-

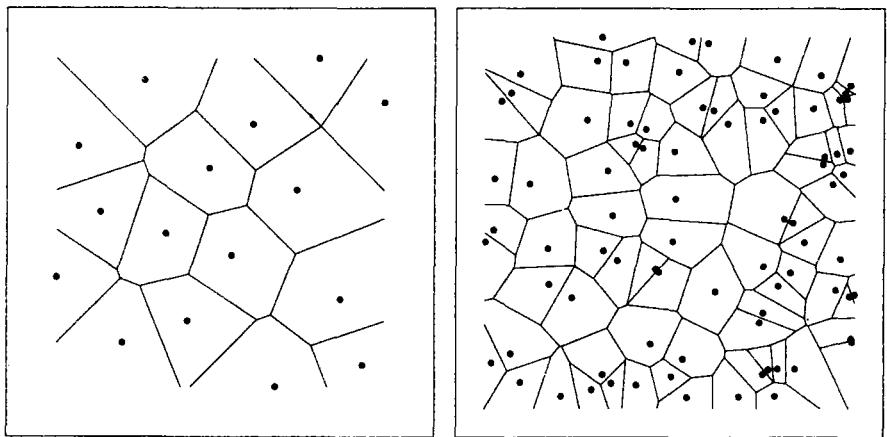


Figura 28.4 Diagrama de Voronoi.

piedades que conducen a algoritmos eficaces para los problemas del punto más cercano. La propiedad que hace eficaces a estos algoritmos es que el número de segmentos de ambos diagramas es proporcional a una pequeña constante multiplicada por N . Por ejemplo, el segmento que conecta los pares de puntos más cercanos debe estar en el dual, lo que significa que se puede resolver el problema de la sección anterior calculando el dual y encontrando simplemente la longitud mínima entre los segmentos del mismo. De forma similar, el segmento que conecta cada punto con su vecino más cercano debe estar en el dual, lo que significa que el problema de todos los vecinos próximos se reduce directamente a encontrar el dual. El cerco convexo del conjunto de puntos es parte del dual,

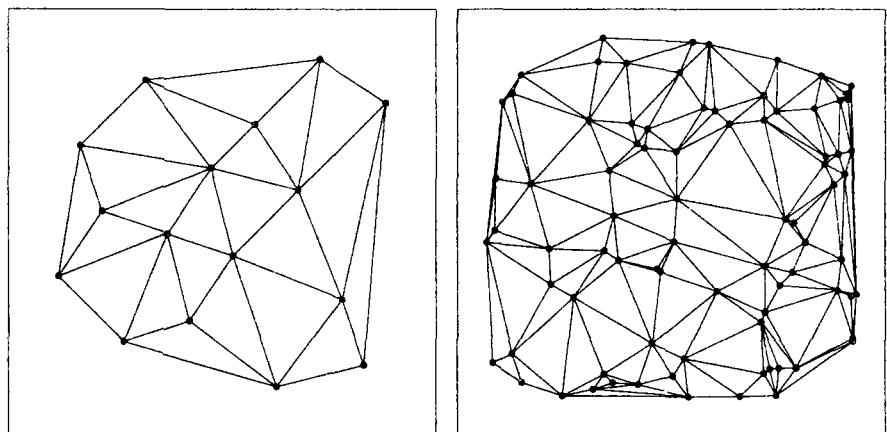


Figura 28.5 Triangulación de Delaunay.

por lo que el cálculo del dual de Voronoi lleva a otro algoritmo de cerco convexo más. En el Capítulo 31 se verá otro ejemplo de un problema que se puede resolver eficazmente encontrando primero el dual de Voronoi.

La propiedad que define al diagrama de Voronoi significa que se puede utilizar para resolver el problema del vecino más próximo: para identificar, en un conjunto de puntos, el vecino más próximo de un punto dado, sólo se necesita encontrar en qué polígono de Voronoi se encuentra el punto. Es posible organizar los polígonos de Voronoi en una estructura como un árbol 2D para permitir que esta búsqueda se haga eficazmente.

El diagrama de Voronoi se puede calcular utilizando un algoritmo con la misma estructura general que el algoritmo anterior del punto más cercano. Primero se ordenan los puntos según su coordenada x . Después se utiliza la ordenación para dividir a los puntos en dos mitades, dejando dos llamadas recursivas para encontrar el diagrama de Voronoi del conjunto de puntos de cada una de las dos mitades. Al mismo tiempo, se ordenan los puntos según y ; finalmente, los diagramas de Voronoi de las dos mitades se fusionan entre sí. Como antes, esta fusión (efectuada cuando pasada vale 2) puede explotar el hecho de que los puntos se ordenan según x antes de las llamadas recursivas y que, después de ellas, se ordenan según y y se construyen los diagramas de Voronoi de las dos mitades. Sin embargo, aun con estas ayudas, la fusión es una tarea bastante complicada y la presentación de una implementación completa rebasa el alcance de este libro.

El diagrama de Voronoi es la estructura natural de los problemas del punto más cercano, y la comprensión de las características de un problema en términos del diagrama de Voronoi o de su dual es sin duda un ejercicio que merece la pena. Sin embargo, para muchos problemas particulares, puede ser conveniente una implementación directa basada en el esquema general dado en este capítulo. Este esquema es lo suficientemente potente para poder calcular el diagrama de Voronoi, por lo que es lo suficientemente potente para algoritmos basados en el diagrama de Voronoi, y puede conducir a programas más simples y eficaces, como se vio para el caso del problema del par más cercano.

Ejercicios

1. Escribir programas para resolver el problema del vecino más próximo, utilizando primero el método de la rejilla y posteriormente árboles 2D.
2. Describir lo que sucede cuando el procedimiento del par más cercano se utiliza en un conjunto de puntos alineados sobre la misma línea horizontal e igualmente espaciados.
3. Describir lo que sucede cuando el procedimiento del par más cercano se utiliza en un conjunto de puntos alineados sobre la misma línea vertical e igualmente espaciados.
4. Dado un conjunto de $2N$ puntos, la mitad con coordenadas positivas de x

y la otra mitad con coordenadas negativas de x , obtener un algoritmo que encuentre el par más cercano constituido por un elemento del mismo en cada mitad.

5. Obtener los pares sucesivos de puntos asignados a $pc1$ y $pc2$ cuando el programa del texto se aplica a los puntos del ejemplo, del que se ha suprimido A.
6. Comprobar la eficacia de atribuir a \min el carácter global comparando el rendimiento de la implementación dada con una implementación puramente recursiva en algún gran conjunto de puntos aleatorios.
7. Obtener un algoritmo para encontrar el par más cercano de un conjunto de segmentos.
8. Dibujar el diagrama de Voronoi y su dual para los puntos A B C D E F del conjunto de puntos del ejemplo.
9. Obtener un método de «fuerza bruta» (que pueda necesitar un tiempo proporcional a N^2) para construir el diagrama de Voronoi.
10. Escribir un programa que utilice la misma estructura recursiva que la implementación del par más cercano dada en el texto para encontrar la superficie convexa de un conjunto de puntos.

REFERENCIAS para los Algoritmos geométricos

En realidad, gran parte del material descrito en esta sección se ha desarrollado hace poco tiempo. Muchos de los problemas y soluciones que se han presentado fueron introducidos por M. Shamos en 1975. La tesis de Ph.D. de Shamos trata un gran número de algoritmos geométricos, que han estimulado muchas de las investigaciones recientes y que finalmente fueron desarrollados en la referencia más autorizada en este campo, el libro de Preparata y Shamos. Esta materia está en rápida expansión: el libro de Edelsbrunner describe muchos de los resultados más recientes.

En su mayor parte, cada algoritmo que se ha presentado está descrito en su propia referencia original. Los algoritmos de cerco convexo del Capítulo 25 se pueden encontrar en los artículos de Jarvis, Graham, y Golin y Sedgewick. Los métodos de búsqueda por rango del Capítulo 26 provienen del artículo de investigación de Bentley y Friedman, que contiene muchas referencias a fuentes originales (de particular interés resulta el artículo original de Bentley sobre los kD árboles, escrito cuando era estudiante). El tratamiento del problema del punto más cercano del Capítulo 28 está basado en el artículo de Shamos y Hoey de 1976, y los algoritmos de intersección geométrica del Capítulo 27 son de su trabajo de 1975 y de un artículo de Bentley y Ottmann.

Pero la mejor vía a seguir por alguien interesado en aprender más sobre algoritmos geométricos consiste en implementar algunos programas y ejecutarlos para aprender sus propiedades y las de los objetos que manipulan.

- J. L. Bentley, «Multidimensional binary search trees used for associative searching», *Communications of the ACM*, **18**, 9 (septiembre, 1975).
- J. L. Bentley y J. H. Friedman, «Data structures for range searching», *Computing Surveys*, **11**, 4 (diciembre, 1979).
- J. L. Bentley y T. Ottmann, «Algorithms for reporting and counting geometric intersections», *IEEE Transactions on Computing*, **C-28**, 9 (septiembre, 1979).
- H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.
- M. Golin y R. Sedgewick, «Analysis of a simple yet efficient convex hull algorithm», *Information Processing Letters*, **1** (1972).
- R. A. Jarvis, «On the identification of the convex hull of a finite set of points in the plane», *Information Processing Letters*, **2** (1973).
- F. P. Preparata y M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.
- M. I. Shamos y D. Hoey, «Closest-point problems» en *16th Annual Symposium on Foundations of Computer Science*, IEEE, 1975.
- M. I. Shamos y D. Hoey, «Geometric intersections problems», en *17th Annual Symposium on Foundations of Computer Science*, IEEE, 1976.

Algoritmos sobre grafos

Algoritmos sobre grafos elementales

Muchos problemas se formulan de manera natural por medio de objetos y de las conexiones entre ellos. Por ejemplo, si se dispone de un mapa de enlaces de líneas aéreas del Este de los Estados Unidos, pueden ser de interés preguntas como: «¿Cuál es el camino más rápido para ir de Providence a Princeton?». O bien puede tener más importancia el precio que el tiempo, y por ello se busca la forma más económica de ir de Providence a Princeton. Para contestar a esta clase de preguntas solamente se necesita tener información sobre las conexiones (líneas aéreas) entre objetos (ciudades).

Los circuitos eléctricos son otro claro ejemplo en el que las conexiones entre objetos tienen un papel principal. Los elementos del circuito, transistores, resistencias y condensadores, están conectados entre sí de forma compleja. Tales circuitos pueden representarse y procesarse por medio de computadoras para poder contestar a preguntas sencillas como «¿Están conectados todos los componentes?», así como a cuestiones más complicadas como «¿Si se construye el circuito, funcionará?». La respuesta a la primera pregunta depende solamente de las propiedades de las conexiones (cables), mientras que la respuesta a la segunda necesita una información detallada sobre las conexiones y los objetos que conectan.

Un tercer ejemplo es la «ordenación de tareas», en el que los objetos son las tareas que se van a realizar, como es el caso de un proceso de fabricación, y las conexiones entre ellos indican qué tareas deben hacerse antes que otras. Aquí el interés se centra en responder a preguntas tales como «¿Cuándo se debe realizar cada tarea?».

Un *grafo* es un objeto matemático que modela fielmente situaciones de este tipo. En este capítulo se examinarán algunas de las propiedades básicas de los grafos, y en los siguientes se estudiará una serie de algoritmos que permitirán responder a preguntas como las propuestas anteriormente.

De hecho, ya se han visto algunos grafos en los capítulos precedentes. Las estructuras de datos enlazadas son realmente representaciones de grafos y algunos de los algoritmos que se verán para el procesamiento de grafos son similares a los que se han visto ya en el tratamiento de árboles y de otras estructuras. Por ejemplo, las máquinas de estados finitos de los Capítulos 19 y 20 se representan por medio de estructuras de grafos.

La teoría de grafos es una rama fundamental de la matemática combinatoria que se ha estudiado en profundidad desde hace cientos de años. Gran parte de las propiedades útiles e importantes de los grafos se han demostrado ya, pero todavía están sin resolver muchos problemas difíciles. En este libro solamente se puede arañar la superficie de lo que se conoce sobre los grafos, abarcando lo suficiente para poder ser capaces de comprender los algoritmos fundamentales.

Como muchos de los temas que se han estudiado en este libro, los grafos no se han examinado desde un punto de vista algorítmico hasta hace poco tiempo. A pesar de que algunos de los algoritmos fundamentales son bastante antiguos, muchos de los más interesantes se han descubierto en los últimos diez años. Incluso los algoritmos triviales conducen a interesantes programas de computadora, y los otros más difíciles que se examinarán posteriormente se encuentran entre los más elegantes e interesantes de los algoritmos conocidos (a pesar de que sean difíciles de comprender).

Glosario

Para el estudio de los grafos hay una cuantiosa cantidad de nomenclatura. La mayor parte de los términos tienen definiciones sencillas, por lo que es conveniente presentarlos todos juntos, aun cuando no se vaya a utilizar algunos de ellos sino hasta más tarde.

Un *grafo* es una colección de *vértices* y de *aristas*. Los vértices son objetos simples que pueden tener un nombre y otras propiedades; una arista es una conexión entre dos vértices. Se puede dibujar un grafo representando los vértices por puntos y las aristas por líneas que los conecten entre sí, pero no hay que olvidar jamás que la definición de un grafo es independiente de la representación. Por ejemplo, los dos dibujos de la Figura 29.1 representan el mismo grafo, que se define diciendo que consiste en el conjunto de vértices A B C D E F G H I J K L M y en el de aristas entre dichos vértices AG AB AC LM JM JL JK ED FD HI FE AF GE.

En algunas aplicaciones, tales como las líneas aéreas del ejemplo anterior, puede que no tenga sentido una reorganización de vértices como la de la Figura 29.1. Pero en otras, como en los mencionados circuitos eléctricos, lo mejor es concentrarse solamente en las aristas y vértices, independientemente de su situación geométrica particular. Y para otras aplicaciones, tales como las máquinas de estados finitos de los Capítulos 19 y 20, no se necesita ninguna disposición geométrica de los nodos. La relación entre los algoritmos sobre grafos y los

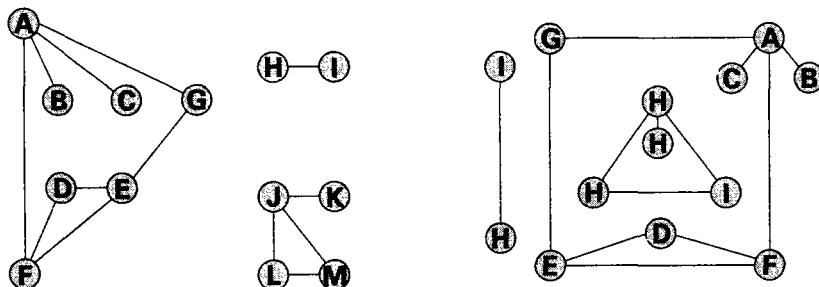


Figura 29.1 Dos representaciones del mismo grafo.

problemas geométricos se presentará con mayor detalle en el Capítulo 31. Por ahora hay que concentrarse en los algoritmos «puros», que tratan colecciones sencillas de aristas y nodos.

Un *camino* entre los vértices x e y de un grafo es una lista de vértices en la que dos elementos sucesivos están conectados por aristas del grafo. Por ejemplo, BAFEG es un camino desde B a G de la Figura 29.1. Un grafo es *conexo* si hay un camino desde cada nodo hacia otro nodo del grafo. De forma intuitiva, si los vértices son objetos físicos y las aristas son cadenas que los conectan, un grafo conexo permanecería en una sola pieza si se le levantara por uno cualquiera de sus vértices. Un grafo que no es conexo está constituido por *componentes conexas*; por ejemplo, el grafo de la Figura 29.1 tiene tres componentes conexas. Un *camino simple* es un camino en el que no se repite ningún vértice (por ejemplo BAFEGAC no es un camino simple). Un *ciclo* es un camino simple con la característica de que el primero y el último vértices son el mismo (un camino desde un punto a sí mismo): el camino AFEGA es un ciclo.

Un grafo sin ciclos se denomina un *árbol* (ver el Capítulo 4). Un grupo de árboles sin conectar se denomina un *bosque*. Un *árbol de expansión* de un grafo es un subgrafo que contiene todos los vértices, pero solamente las aristas necesarias para formar un árbol. Por ejemplo, las aristas AB AC AF FD EG ED forman un árbol de expansión de la componente mayor del grafo de la Figura 29.1, y la Figura 29.2 muestra un grafo más grande, así como uno de sus árboles de expansión.

Hay que subrayar que si se añade una arista cualquiera a un árbol, se debe formar un ciclo (dado que ya existe un camino entre los dos vértices que ella conecta). Además, como se vio en el Capítulo 4, un árbol con V vértices tiene exactamente $V-1$ aristas. Si un grafo con V vértices tiene menos de $V-1$ aristas, no puede ser conexo. Si tiene más de $V-1$ aristas, debe contener un ciclo. (Pero si tiene exactamente $V-1$ aristas no es necesariamente un árbol.)

En este libro se denominará V al número de vértices que tiene un grafo y A al de aristas. Es de destacar que A puede estar comprendido entre 0 y $\frac{1}{2}V(V - 1)$. Los grafos con todas las aristas posibles se denominan grafos *completos*; los que

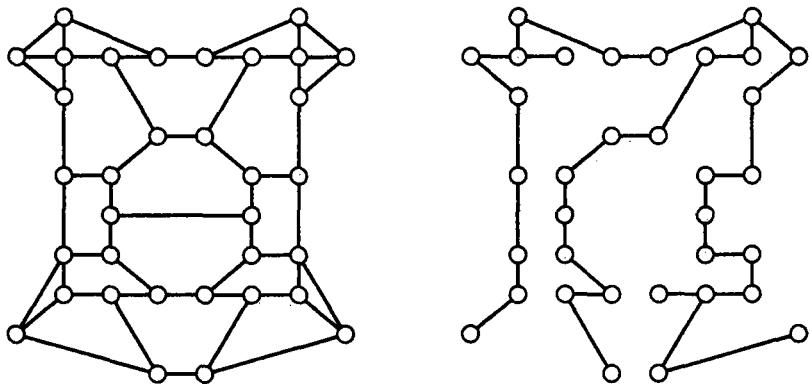


Figura 29.2 Un grafo muy grande y uno de sus árboles de expansión.

tienen relativamente pocas (menos de $V \log V$) se denominan *dispersos* y a los que les faltan muy pocas de todas las posibles se les denomina *densos*.

El hecho de que la topología de los grafos dependa fundamentalmente de dos parámetros hace que el estudio comparativo de los algoritmos sobre grafos sea algo más complicado que el de muchos de los algoritmos que se han estudiado, ya que aparecen más posibilidades. Por ejemplo, un algoritmo puede necesitar V^2 pasos, mientras que otro, para el mismo problema, puede necesitar $(A + V)\log A$ pasos. El segundo algoritmo sería preferible para grafos dispersos y el primero para grafos densos.

Los grafos que se han descrito hasta ahora son del tipo más sencillo, el denominado de *grafos no dirigidos*. También se consideran en este libro otros tipos de grafos más complicados, en los que se asocia más información con los nodos y las aristas. En los *grafos ponderados* se asignan enteros (*pesos*) a cada arista para representar, por ejemplo, distancias o costes. En los *grafos dirigidos*, las aristas son de «sentido único»: una arista puede ir de x a y pero no de y a x . Los grafos dirigidos ponderados se denominan a veces *redes*. Como se verá posteriormente, la información extra que contienen los grafos ponderados y dirigidos hace que éstos sean algo más difíciles de manipular que los grafos no dirigidos sencillos.

Representación

Con el fin de procesar grafos por medio de un programa de computadora se necesita decidir cómo representarlos en la máquina. Aquí se estudiarán dos de las representaciones más usuales; la elección entre ellas dependerá normalmente de

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	1	0	0
L	0	0	0	0	0	0	0	0	0	1	0	1	1
M	0	0	0	0	0	0	0	0	0	1	0	1	1

Figura 29.3 Representación por matriz de adyacencia.

si el grafo es denso o disperso, aunque, como siempre, la naturaleza de la operación a realizar también tendrá un papel importante.

El primer paso para representar un grafo es hacer corresponder los nombres de los vértices con los enteros entre 1 y V . La principal razón para hacer esto es facilitar un rápido acceso a la información que corresponde a cada vértice, utilizando un array indexado. Para este objetivo puede utilizarse cualquier esquema estándar de búsqueda; por ejemplo, se pueden transformar los nombres de los vértices en enteros entre el 1 y el V por medio de una tabla de dispersión o de un árbol binario donde se pueda buscar el entero correspondiente al nombre de un vértice cualquiera. Como ya se han estudiado estas técnicas, se supone que existe una función índice para convertir nombres de vértices en enteros entre 1 y V , y una función nombre para convertir enteros en nombres de vértices. Para simplificar los algoritmos se utilizarán nombres de vértices de una sola letra, correspondiendo la i -ésima letra del alfabeto al entero i . Así, aunque nombre e índice son de fácil implementación en los ejemplos, su utilización hará más sencilla la extensión de los algoritmos a la manipulación de grafos con nombres de vértices reales, utilizando las técnicas de los Capítulos 14-17.

La representación más directa de los grafos es la denominada representación por *matriz de adyacencia*. Se construye un array de V^2 valores booleanos en el que $a[x][y]$ es igual a 1 si existe una arista desde el vértice x al y y a 0 en el caso contrario. La matriz de adyacencia del grafo de la Figura 29.1 se muestra en la Figura 29.3.

Es de destacar que en realidad cada arista se representa con dos bits: una arista que enlace x e y se representa con valores verdaderos tanto en $a[x][y]$ como en $a[y][x]$. Aunque sea posible ahorrar espacio almacenando solamente

la mitad de esta matriz simétrica, en C++ no es conveniente hacer esto, y los algoritmos son algo más simples con la matriz completa. Además, normalmente se supone que existe una «arista» desde cada vértice a sí mismo, por lo que $a[x][x]$ es igual a 1 para los valores de x desde 1 hasta V . (En algunos casos es más conveniente poner a 0 los elementos de la diagonal; en el libro se hará esto libremente cuando se considere apropiado.)

Un grafo se define por un conjunto de nodos y otro de aristas que los conectan. Para aceptar un grafo como entrada se necesita establecer un formato de lectura de estos dos conjuntos. Una posibilidad consiste en utilizar para ello la propia matriz de adyacencia, pero, como se verá, esto no es adecuado para los grafos densos. Por ello se utilizará un formato más directo: primero se leen los nombres de los vértices, después los pares de nombres de vértices (lo que define las aristas). Como se mencionó anteriormente, una sencilla forma de actuar es leer los nombres de los vértices en una tabla de dispersión o en un árbol binario de búsqueda, y asignar un entero a cada nombre de vértice, que servirá para poder indexar por vértices a los arrays de igual forma que en la matriz de adyacencia. El i -ésimo vértice leído puede asignarse al entero i . Para simplificar más los programas, se leen primero V y A , a continuación los vértices y después las aristas. Alternativamente, se podría distribuir la entrada por medio de un delimitador que separe los vértices de las aristas, y el programa podría determinar V y A a partir de los datos de entrada. (En los ejemplos anteriores se utilizan las V primeras letras del alfabeto como nombres de vértices, lo que permite simplificar el esquema leyendo V y A , y después los A pares de letras de las primeras V letras del alfabeto.) El orden en el que aparecen las aristas no tiene ninguna importancia dado que todas las permutaciones de las aristas representan el mismo grafo y generan la misma matriz de adyacencia, como muestra el siguiente programa:

```

int V, A;
int a[maxV][maxV];
void matrizady()
{
    int j, x, y;
    cin >> V >> A;
    for (x = 1; x <= V; x++)
        for (y = 1; y <= V; y++) a[x][y] = 0;
    for (x = 1; x <= V; x++) a[x][x] = 1;
    for (j = 1; j <= A; j++)
    {
        cin >> v1 >> v2;
        x = indice(v1); y = indice(v2);
        a[x][y] = 1; a[y][x] = 1;
    }
}

```

En este programa se omiten tanto el tipo de v_1 y v_2 como el código de índice. Estos detalles se pueden añadir de manera sencilla, dependiendo de la representación que se desee para el grafo de entrada. Para los ejemplos del libro, v_1 y v_2 pueden ser del tipo char e índice podría ser una simple función que devuelva $c - 'A' + 1$ o algo similar.

Se puede desarrollar fácilmente una clase de C++ para grafos que permita ocultar la representación detrás de una interfaz que consista en funciones básicas para aplicar a los grafos. Aunque se ha demostrado esta metodología para estructuras ampliamente utilizadas, como diccionarios y colas de prioridad, se debe evitar hacerlo en los algoritmos sobre grafos, dado que las implementaciones que utilizan variables globales son algo más compactas, y porque las implementaciones que dependen de la aplicación tienden a ser necesarias en una buena implementación de clase. Inicialmente el objetivo de los algoritmos sobre grafos es exponer las diferencias de las representaciones, no ocultarlas. Por ello el lector más experto puede elegir una representación adecuada y utilizar las capacidades de abstracción de datos de C++ como ayuda para integrarla en una aplicación particular.

La representación por matriz de adyacencia sólo es satisfactoria si los grafos a procesar son densos: la matriz necesita V^2 bits de almacenamiento y V^2 pasos de inicialización. Si el número de aristas (el número de bits 1 de la matriz) es proporcional a V^2 , se puede aceptar esta representación porque en cualquier caso se necesitan aproximadamente V^2 pasos para leer las aristas. Sin embargo, si el grafo es disperso, la simple inicialización de la matriz podría ser el factor dominante en el tiempo de ejecución del algoritmo. Ésta podría ser también la mejor representación para algunos algoritmos cuya ejecución necesita más de V^2 pasos.

A continuación se estudia una representación mejor adaptada a los casos de grafos que no son densos. En la representación por *estructura de adyacencia* todos los vértices conectados con uno dado se relacionan en una *lista de adyacencia* de dicho vértice. Esto se puede realizar fácilmente por medio de listas enlazadas, como se muestra en el siguiente programa que construye la estructura de adyacencia para el grafo del ejemplo. Las listas enlazadas se construyen de la forma habitual, con un nodo ficticio z en cola (apuntando sobre sí mismo). Los nodos ficticios del encabezamiento de las listas se conservan en un array ady indexado por vértices. Para añadir una arista que conecte x a y en esta representación del grafo, se agregará x a la lista de adyacencia de y e y a la lista de adyacencia de x :

```
struct nodo
    { int v; struct nodo *siguiente; };
int V, A;
struct nodo *ady[maxV], *z;
void listaady()
{
```

```

int j, x, y; struct nodo *t;
cin >> V >>A;
z = new nodo; z->siguiente = z;
for (j = 1; j <= V; j++) ady[j] = z;
for (j = 1; j <= A; j++)
{
    cin >> v1 >> v2;
    x = indice(v1); y = indice(v2);
    t = new nodo;
    t->v =x; t->siguiente = ady[y]; ady[y]=t;
    t = new nodo;
    t->v =y; t->siguiente = ady[x]; ady[x]=t;
}
}

```

La representación por lista de adyacencia es la mejor para los grafos dispersos, dado que el espacio que se necesita está en $O(V + A)$, en contraste con el espacio en $O(V^2)$ necesario para la representación por matriz de adyacencia.

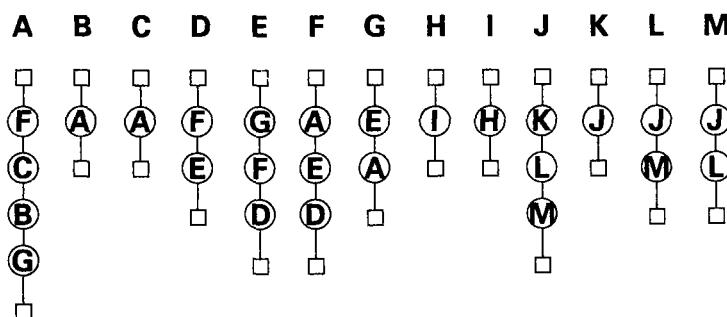


Figura 29.4 Una representación por estructura de adyacencia.

Si las aristas aparecen en el orden AG AB AC LM JM JL JK ED FD HI FE AF GE, el programa anterior construye la estructura de listas de adyacencia que se muestra en la Figura 29.4. Se observa otra vez que cada arista se representa dos veces: una arista que conecte x e y se representa como un nodo que contiene a x en la lista de adyacencia de y , y como un nodo que contiene a y en la lista de adyacencia de x . Es importante incluir ambos nodos, pues en caso contrario una pregunta tan simple como «¿Qué nodos están conectados directamente al nodo x ?» no podría contestarse de forma eficaz.

En esta representación el orden en el que aparecen las aristas en la entrada es muy importante: él determina (junto con el método de inserción utilizado) el orden en el que aparecerán los vértices en las listas de adyacencia. Por ello el

el mismo grafo se puede representar de muchas formas diferentes en una estructura de listas de adyacencia. De hecho, es difícil predecir que las listas de adyacencia serán semejantes al examinar solamente la secuencia de aristas, dado que cada arista implica la inserción en dos listas de adyacencia.

El orden de aparición de las aristas en la lista de adyacencia afecta, a su vez, el orden en el que serán procesadas por los algoritmos. Esto es, la estructura de la lista de adyacencia determina la forma en la que diversos algoritmos «verán» al grafo. Mientras que un algoritmo debe dar una respuesta correcta, sin que tenga importancia cómo están ordenadas las aristas en las listas de adyacencia, podría ser que obtuviera esta respuesta por muchas secuencias de cálculo distintas en órdenes diferentes. Y si existe más de una «respuesta correcta», diferentes órdenes de entrada podrían llevar a resultados de salida diferentes.

En esta representación no se contemplan algunas operaciones simples. Por ejemplo, se podría desear suprimir un vértice, x , y todas las aristas que inciden en él. No es suficiente con suprimir nodos de una lista de adyacencia: cada nodo de la lista especifica otro vértice, cuya lista de adyacencia debe buscarse para eliminar un nodo correspondiente a x . Este problema puede corregirse enlazando los dos nodos de las listas que corresponden a una arista determinada y haciendo las listas de adyacencia doblemente enlazadas. Así, si se suprime una arista, los dos nodos de las listas que se corresponden con ella pueden eliminarse rápidamente. Por supuesto, estos lazos extra son incómodos para el proceso y no se les debería incluir salvo que se necesitaran operaciones como la de eliminación.

Estas consideraciones también justifican por qué no se utilizan representaciones «directas» en los grafos: una estructura de datos que modela el grafo exactamente, con los vértices representados por registros asignados y listas de aristas que contienen enlaces a los vértices en lugar de nombres de vértices. Sin acceso directo a los vértices, las operaciones más simples podrían convertirse en verdaderos desafíos. Por ejemplo, para añadir una arista a un grafo representado de esta manera se tendría que buscar a través del grafo alguna forma de encontrar los vértices.

Los grafos dirigidos y los grafos ponderados se representan por medio de estructuras similares. En el caso de los grafos dirigidos todo lo expuesto es válido excepto que cada arista se representa una sola vez: una arista de x a y se representa con 1 en $a[x][y]$ de la matriz de adyacencia o por la aparición de y en la lista de adyacencia de x de la estructura de adyacencia. Así se puede representar un grafo no dirigido como un grafo dirigido en el que toda arista que conecta dos vértices es una arista dirigida en los dos sentidos. Para los grafos ponderados se procede exactamente igual excepto que se completa la matriz de adyacencia con pesos en lugar de valores booleanos (utilizando algún peso que no exista para representar la ausencia de una arista), o incluyendo un campo para el peso en los registros de la lista de la estructura de adyacencia.

A veces es necesario asociar otras informaciones a los vértices o nodos de un grafo para permitir el modelado de objetos más complicados o para ahorrar trabajo en la actualización de la información de los algoritmos complicados. Para

disponer de esta información extra asociada con cada vértice se pueden utilizar arrays auxiliares indexados por los números de los vértices o transformando *ady* en un array de registros en la representación de la estructura de adyacencia. La información suplementaria asociada con cada arista puede colocarse en los nodos de la lista de adyacencia (o en un array *a* de registros en la representación por matriz de adyacencia), o en arrays auxiliares indexados por el número de arista (lo que requiere numerarlas).

Búsqueda en profundidad

Al comienzo de este capítulo, se han visto varias cuestiones que aparecen de forma inmediata cuando se procesa un grafo. ¿El grafo es conexo? Si no lo es, ¿cuáles son sus componentes conexas? ¿Contiene un ciclo? Estos problemas y otros muchos pueden solucionarse fácilmente por medio de una técnica denominada *búsqueda en profundidad*, que es un medio natural de «explorar» cada nodo y de comprobar cada arista del grafo de forma sistemática. En los capítulos siguientes se verá que es posible utilizar sencillas variaciones de una generalización de este método para resolver una gran variedad de problemas sobre grafos.

Por ahora hay que concentrarse en los mecanismos que examinan metódicamente cada elemento del grafo. Se utiliza un array *val*[*V*] para registrar el orden en el que se exploran los vértices. Cada entrada del array se inicializa con el valor *novisto* para indicar qué vértices no se han inspeccionado todavía. El objetivo es visitar sistemáticamente todos los vértices del grafo, colocando el orden del vértice explorado, *id*, en la *id*-ésima entrada de *val*, para los valores de *id*= 1, 2, ..., *V*. El siguiente programa utiliza un procedimiento *visitar* que inspecciona todos los vértices de la misma componente conexa del vértice pasado como argumento.

```
void buscar()
{
    int k;
    for (k=1; k<=V; k++) val[k] = novisto;
    for (k=1; k<=V; k++)
        if (val[k] == novisto); visitar(k);
}
```

El primer bucle *for* inicializa el array *val*. A continuación se invoca *visitar* para el primer vértice, con el resultado de atribuir valores en *val* a todos los vértices conectados a él. A continuación *buscar* explora el array *val* buscando vértices que todavía no hayan sido vistos y llama a *visitar* para estos vértices, continuando de esta forma hasta que se hayan inspeccionado todos los vértices.

Es de destacar que este método no depende de la forma como se represente el grafo o de como se implemente visitar.

En primer lugar se considera una implementación recursiva de visitar para la representación por listas de adyacencia: para visitar un vértice, se comprobarán todas sus aristas para ver si conducen a vértices que todavía no se han visto; si los hay, se invoca visitar para ellos.

```
void visitar(int k) // BP, listas de adyacencia
{
    struct nodo *t;
    val[k] = ++id;
    for (t = ady[k]; t != z; t = t->siguiente)
        if (val[t->v] == novisto) visitar(t->v);
}
```

La Figura 29.5 traza el recorrido de la operación de búsqueda en profundidad de la componente mayor del grafo del ejemplo y muestra cómo se toca cada arista de esta componente como resultado de la llamada visitar(1) (después de que se hayan construido las listas de adyacencia de la Figura 29.4). Realmente se «contacta» *dos veces* con cada arista, dado que todas están representadas en las dos listas de adyacencia de los vértices que conectan. En la Figura 29.5 hay un diagrama por cada arista recorrida (cada vez que el enlace *t* se pone a apuntar a algún nodo de alguna lista de adyacencia). En cada diagrama la arista «actual» aparece sombreada y el nodo cuya lista de adyacencia contiene a esta arista está etiquetado con un cuadrado. Además, cada vez que se inspecciona un nodo por primera vez (lo que se corresponde con una nueva llamada a visitar), se representa en negro la arista que conduce a dicho nodo. Los nodos que no han sido tocados todavía están sombreados, pero no etiquetados, y aquellos para los que visitar ha terminado están sombreados y etiquetados.

La primera arista recorrida es AF, el primer nodo de la primera lista de adyacencia. A continuación se invoca visitar para el nodo F y se recorre la arista FA, dado que A es el primer nodo de la lista de adyacencia de F. Pero el nodo A es en este momento novisto, por lo que se coge la arista FE, la siguiente entrada de la lista de adyacencia de F. A continuación se recorre EG y después GE, dado que G y E son los primeros de cada una de las otras listas. Luego se recorre GA y con esto se termina visitar G, por lo que el algoritmo continua con visitar E y recorre EF y después ED. Después visitar D consiste en recorrer DE y DF, ninguna de las cuales conduce a un nuevo nodo. Dado que D es el último nodo de la lista de adyacencia de E, visitar este nodo ha terminado y la visita de F se completará recorriendo FD. Finalmente se vuelve a A y se recorre AC, CA, AB, BA y AG.

Otra forma de seguir la operación de búsqueda en profundidad es volver a dibujar el grafo en el orden indicado por las llamadas recursivas del procedimiento visitar, como se muestra en la Figura 29.6. Cada componente conexa

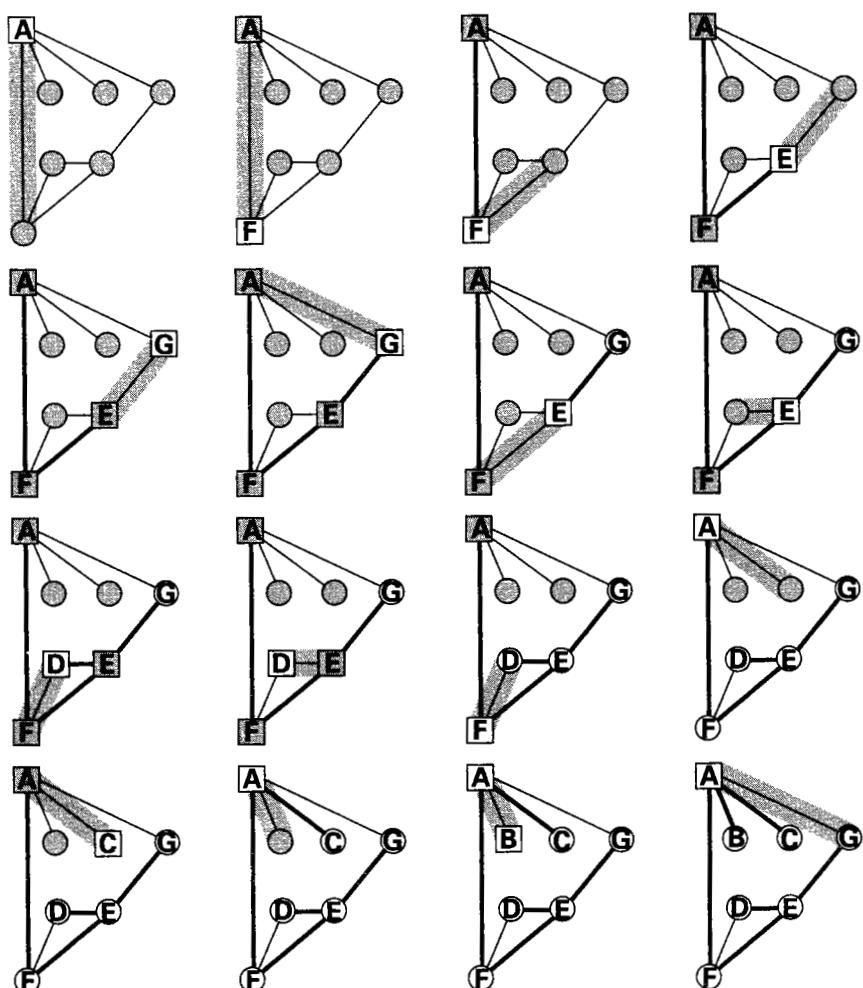


Figura 29.5 Búsqueda en profundidad (recursiva) de la componente mayor del grafo.

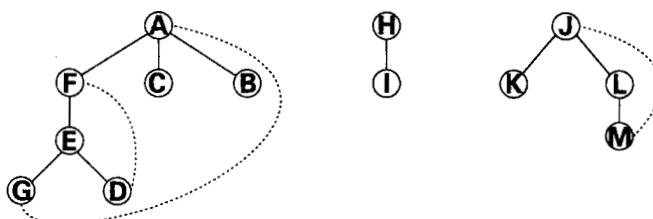


Figura 29.6 Bosque de búsqueda en profundidad.

conduce a un árbol, denominado *árbol de búsqueda en profundidad* de la componente. Recorriendo este árbol en orden previo se obtienen los vértices del grafo en el orden en el que estaban la primera vez que se encontraron en la búsqueda; recorriéndolo en orden posterior se obtienen los vértices en el orden que estaban al terminar visitar. Es importante comprender que este bosque de árboles de búsqueda en profundidad es simplemente otra forma de dibujar el grafo: el algoritmo examina todos los vértices y aristas del grafo.

Las líneas de trazo grueso de la Figura 29.6 indican que el algoritmo ha encontrado al vértice inferior en la lista de aristas del vértice superior y no había sido inspeccionado hasta ahora, por lo que se hizo una llamada recursiva. Las líneas de puntos corresponden a las aristas dirigidas hacia vértices que ya han sido visitados, por lo que la comprobación *if de visitar* ha fallado y la arista no ha «provocado» una llamada recursiva. Estos comentarios se aplican la *primera* vez que se encuentra a cada arista; la comprobación *if de visitar* permite también evitar que se recorra la arista la *segunda* vez que se la encuentre, como se vio en la Figura 29.5.

Una propiedad crucial de estos árboles de búsqueda en profundidad para grafos no dirigidos es que los enlaces de puntos siempre van desde un nodo a algún *antecesor* (otro nodo del mismo árbol situado más alto en el camino hacia la raíz). En cualquier momento de la ejecución del algoritmo los vértices se dividen en tres clases: aquellos para los que *visitar* ha terminado, aquellos para los que ha terminado sólo parcialmente y aquellos que todavía no han sido encontrados. Por la definición de *visitar* no se podrá encontrar jamás una arista apuntando hacia un vértice de la primera clase y si se encuentra una dirigida hacia un vértice de la tercera clase, se efectuará una llamada recursiva (por lo que la arista se representará con una línea gruesa en el árbol de búsqueda en profundidad). Los únicos vértices que quedan son los de la segunda clase, que son precisamente los situados en el camino desde el vértice actual al de la raíz del mismo árbol, y toda arista dirigida hacia uno cualquiera de ellos corresponderá a un enlace de puntos del árbol de búsqueda en profundidad.

Propiedad 29.1 *La búsqueda en profundidad de un grafo representado con listas de adyacencia necesita un tiempo proporcional a $V + A$.*

Se actualiza cada uno de los V valores de *val* (de ahí el término V) y se examina cada arista dos veces (de ahí el término A). Se podría encontrar un grafo (extremadamente) denso con $A < V$, pero si no están permitidos los vértices aislados (se podría, por ejemplo, haberlos suprimido en una fase de preprocesamiento), es preferible pensar que el tiempo de ejecución de la búsqueda en profundidad es lineal en el número de aristas.■

El mismo método básico se puede aplicar a los grafos representados con matrices de adyacencia, utilizando el procedimiento *visitar* siguiente:

```
void visitar(int k) // BP, matriz de adyacencia
```

```

{
int t;
val[k] = ++id;
for (t = 1; t <= V; t++)
    if (a[k][t] != 0)
        if (val[t] == novisto) visitar(t);
}

```

El recorrido a través de una lista de adyacencia se traduce en una exploración de las filas de la matriz de adyacencia, buscando valores iguales a 1 (que corresponden a las aristas). Como anteriormente, cualquier arista dirigida hacia un vértice que todavía no ha sido visto se «recorre» por medio de una llamada recursiva. Ahora, las aristas conectadas a cada vértice se examinarán en un orden diferente, lo que proporciona un bosque de búsqueda en profundidad diferente, representado en la Figura 29.7. Esto confirma el hecho de que un bosque de búsqueda en profundidad no es más que otra representación del grafo, cuya estructura particular depende a la vez del algoritmo de búsqueda y de la representación interna utilizada.

Propiedad 29.2 *La búsqueda en profundidad de un grafo representado con una matriz de adyacencia necesita un tiempo proporcional a V^2 .*

La demostración de esta propiedad es trivial: se comprueba todo bit de la matriz de adyacencia. ■

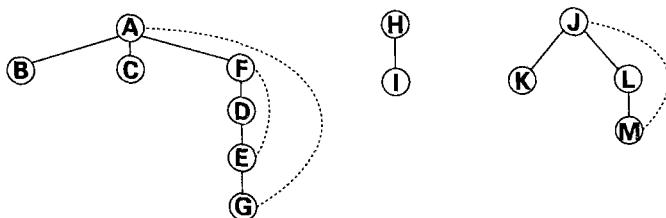


Figura 29.7 Bosque de búsqueda en profundidad (representación del grafo por matriz).

La búsqueda en profundidad resuelve directamente algunos problemas elementales de procesamiento de grafos. Por ejemplo, el procedimiento se basa en encontrar las sucesivas componentes conexas: el número de componentes conexas es igual al número de veces que se llama a visitar en la última línea del programa. El comprobar si un grafo tiene un ciclo es también una modificación trivial del programa anterior. Un grafo tiene un ciclo si, y sólo si, se descubre un nodo no visto en visitar. Esto es, si se encuentra una arista que apunta a un vértice que ya se ha inspeccionado, entonces se tiene un ciclo. De forma

equivalente, todos los enlaces en línea punteada de los árboles de búsqueda en profundidad pertenecen a ciclos.

Búsqueda en profundidad no recursiva

La búsqueda en profundidad de un grafo es una generalización del recorrido de árboles. Si el grafo es un árbol, es exactamente equivalente al recorrido del mismo; para los grafos, corresponde al recorrido del árbol que recubre al grafo y que se «descubre» durante el proceso de búsqueda. Como se ha visto, el árbol a recorrer depende de la forma como se representa el grafo.

Se puede eliminar la recursión en la búsqueda en profundidad utilizando una pila, de la misma forma que se hizo para el recorrido del árbol del Capítulo 5. Para los árboles se encontró que la supresión de la recursión conducía a una implementación equivalente alternativa (relativamente simple) y también se descubrió un algoritmo de recorrido no recursivo (por niveles). Para los grafos se encontrará una evolución similar, que finalmente conducirá (en el Capítulo 31) a un algoritmo de recorrido de grafos de uso general.

Sirviéndose de la experiencia del Capítulo 5, se puede dar directamente una implementación basada en una pila:

```
Pila pila(maxV);
void visitar(int k) //BP no recursiva, listas de adyacencia
{
    struct nodo *t;
    pila.meter(k);
    while (!pila.vacia())
    {
        k = pila.sacar(); val[k] = ++id;
        for (t = ady[k]; t != z; t = t->siguiente)
            if (val[t->v] == novisto)
                { pila.meter(t->v); val[t->v] = —1; }
    }
}
```

Los vértices que han sido tocados, pero que todavía no han sido inspeccionados, se colocan en una pila. Para visitar a un vértice se recorren sus aristas y se mete en la pila cualquier vértice que no haya sido inspeccionado todavía y que no esté todavía en la pila. En la implementación recursiva, la «contabilidad» de los vértices «parcialmente inspeccionados» está oculta por la variable local *t* del procedimiento recursivo. Habría sido posible implementar esto directamente guardando los punteros (correspondientes a *t*) en los elementos de las listas de adyacencia, y así sucesivamente. En su lugar se ha extendido simplemente el

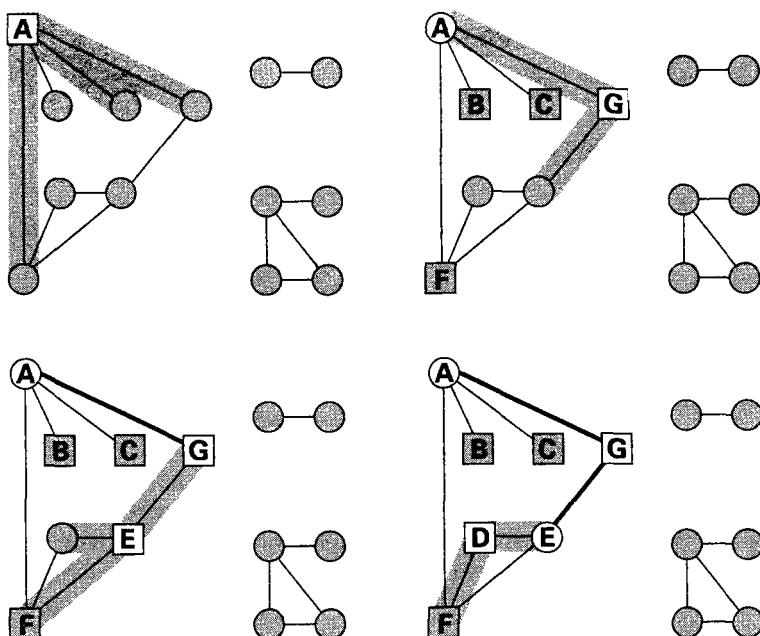


Figura 29.8 Comienzo de la búsqueda basada en una pila.

significado de las entradas de `val` para englobar a los vértices que ya están en la pila: los vértices con entradas de `val` iguales a `novisto` no se han encontrado todavía (como antes), aquellos con entradas negativas de `val` están en la pila y aquellos con entradas de `val` entre 1 y V ya han sido explorados (todas las aristas de sus listas de adyacencia se han puesto en la pila).

La Figura 29.8 representa la operación de este procedimiento de búsqueda en profundidad basado en pilas cuando se han inspeccionado los cuatro primeros nodos del grafo del ejemplo. Cada diagrama de esta figura corresponde a la inspección de un nodo: el nodo visitado se dibuja como un cuadrado y todas las aristas de sus listas de adyacencia están sombreadas. Como antes, los nodos que no se han encontrado todavía no tienen etiqueta y están sombreados, los nodos para los que la exploración ha terminado están etiquetados y no sombreados, y cada nodo está conectado por una arista en línea negra gruesa al nodo que ha provocado que se coloque en la pila. Los nodos que están todavía en la pila están dibujados con cuadrados.

El primer nodo explorado es A: se recorren las aristas AF, AB, AC y AG y se colocan en la pila F, B, C y G. A continuación se saca de la pila a G (es el *último* nodo de la lista de adyacencia de A) y se recorren las aristas GA y GE, lo que se traduce en colocar E en la pila (A no está todavía en ella). Después se recorren EG, EF y ED y se mete a D en la pila, etc. La Figura 29.9 muestra el

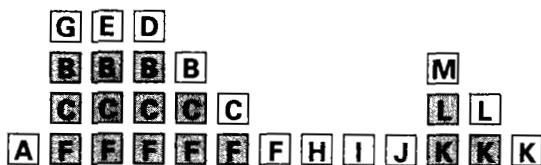


Figura 29.9 Contenido de la pila durante la búsqueda basada en una pila.

contenido de la pila durante la búsqueda y la Figura 29.10 muestra cómo continúa el proceso de la Figura 29.8.

El lector seguramente habrá notado que el programa precedente *no* inspecciona las aristas y nodos en el mismo orden que en la implementación recursiva. Esto podría hacerse colocando las aristas en la pila en orden inverso y manipulando de forma diferente el caso en el que se encuentra de nuevo a un nodo que ya está en la pila. En primer lugar, si se colocan en la pila las aristas de la lista de adyacencia de cada nodo en orden inverso al que aparecen en la lista,

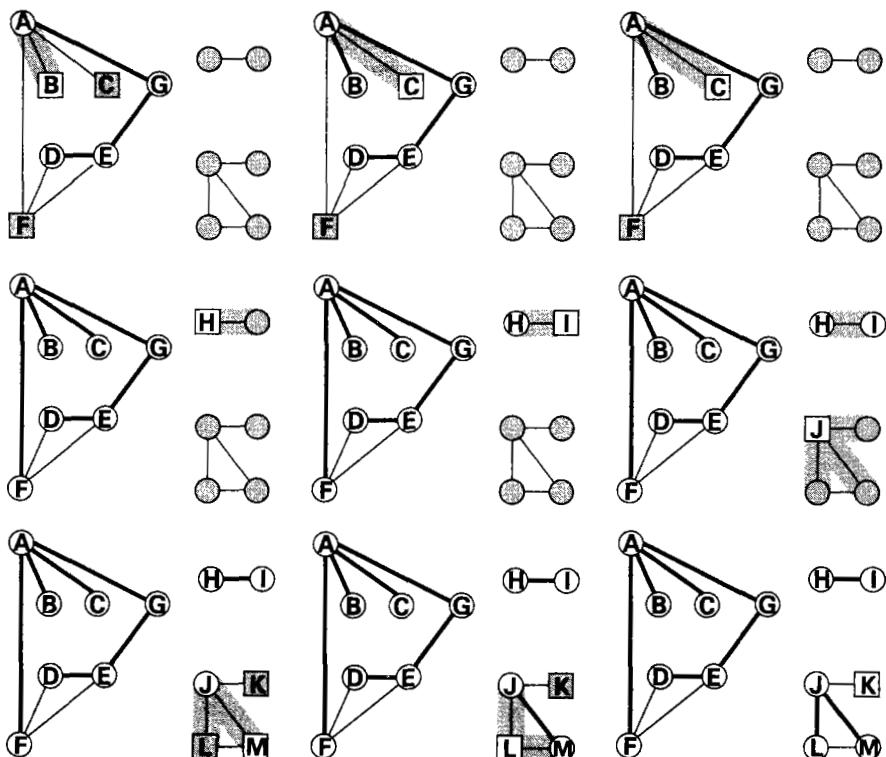


Figura 29.10 Fin de la búsqueda basada en una pila.

entonces se sacarán de la pila y se visitarán los nodos correspondientes en el mismo orden que en la implementación recursiva. (Éste es el mismo efecto que en el recorrido del árbol del ejemplo del Capítulo 5, en el que el subárbol derecho se coloca en la pila antes que el izquierdo en la implementación no recursiva.) Una diferencia más importante es que el método basado en la pila técnicamente no es del todo un procedimiento de «búsqueda en profundidad», dado que inspecciona el último nodo que se ha colocado en la pila, no el último nodo que se ha encontrado, como es el caso de la búsqueda en profundidad recursiva. Esto se puede restablecer moviendo hacia la parte superior de la pila los nodos que están en ella, según se van redescubriendo, pero esta operación necesita una estructura de datos más sofisticada que una pila. En el Capítulo 31 se examinará una forma más simple de implementar esto.

Búsqueda en amplitud

De igual forma que en el recorrido del árbol (ver Capítulo 4), se puede utilizar una *cola*, en vez de una pila, como estructura de datos para almacenar vértices. Esto conduce a un segundo algoritmo clásico de recorrido de grafos, denominado *búsqueda en amplitud*. Para implementar la búsqueda en amplitud, se cambian las operaciones de pila por operaciones de cola en el programa de búsqueda anterior:

```
Cola cola(maxV);
void visitar(int k) // BA, listas de adyacencia
{
    struct nodo *t;
    cola.poner(k);
    while (!cola.vacia())
    {
        k = cola.obtener();
        l = ++id;
        for (t = ady[k]; t != z; t = t->siguiente)
            if (val[t->v] == novisto)
                { cola.poner(t->v); val[t->v] = -1; }
    }
}
```

El hecho de cambiar la estructura de datos de esta forma afecta el orden de inspección de los nodos. En el grafo pequeño del ejemplo las aristas se visitan en el orden AF AC AB AG FA FE FD CA BA GE GA DF DE EG EF ED HI IH JK JL JM KJ LJ LM MJ ML. El contenido de la cola durante el recorrido se muestra en la Figura 29.11.

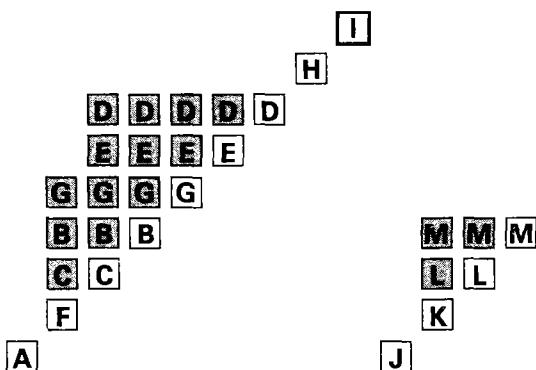


Figura 29.11 Contenido de la cola durante la búsqueda en amplitud.

Como en la búsqueda en profundidad, se puede definir un bosque a partir de las aristas que conducen por primera vez a cada nodo, como se muestra en la Figura 29.12. La búsqueda en amplitud corresponde a recorrer los árboles de este bosque por orden de niveles.

En los dos algoritmos se puede imaginar que los vértices están divididos en tres clases: vértices del *árbol* (o *visitados*), aquellos que se han retirado de la estructura de datos; vértices del *margen*, que son adyacentes a los vértices del árbol, pero que no se han inspeccionado todavía, y vértices *no vistos*, que no se han encontrado todavía. Si cada vértice del árbol está conectado a la arista que ha provocado su inserción en la estructura de datos (las aristas de trazo negro grueso de las Figuras 29.8 y 29.10), entonces estas aristas forman un árbol.

Para buscar de forma sistemática una componente conexa de un grafo (implementando un procedimiento visitar), se comienza por un vértice del margen, siendo todos los otros no vistos, y se lleva a cabo la acción siguiente hasta que se hayan inspeccionado todos los vértices: «se transporta un vértice x desde el margen al árbol, y se colocan en el margen todos los vértices no vistos adyacentes a x ». Los métodos de recorrer los grafos difieren en la forma como deciden qué vértice debe pasar desde el margen hacia el árbol. En la búsqueda en pro-

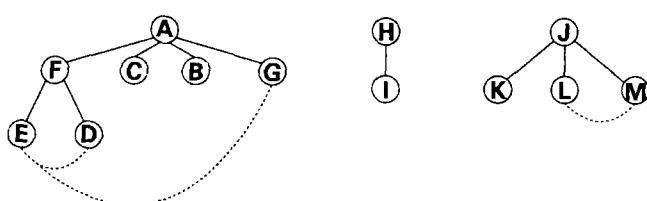


Figura 29.12 Bosque de búsqueda en amplitud.

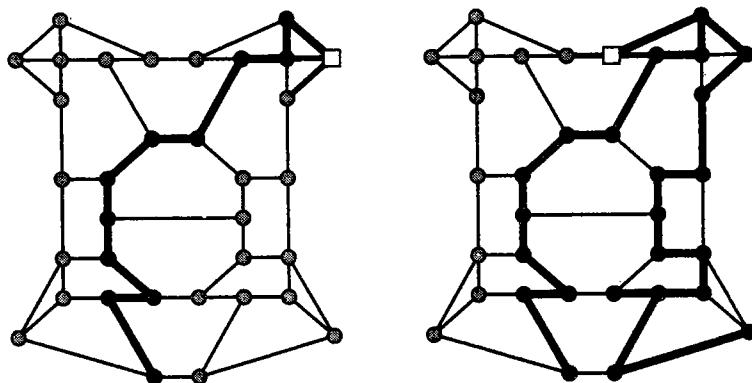


Figura 29.13 Búsqueda en profundidad en un gran grafo.

fundidad se desea elegir el vértice del margen que se ha encontrado más recientemente; esto corresponde al empleo de una pila para almacenar los vértices del margen. En la búsqueda en amplitud se desea elegir el vértice del margen que se ha encontrado *menos* recientemente; esto corresponde al empleo de una cola para almacenar los vértices del margen. En el Capítulo 31 se verá el efecto de utilizar una *cola de prioridad* para el margen.

El contraste entre las búsquedas en profundidad y en amplitud se hace más evidente cuando se considera un gran grafo. La Figura 29.13 muestra el proceso de búsqueda en profundidad en un gran grafo, al tercio y a los dos tercios de su realización; la Figura 29.14 es la descripción correspondiente a la búsqueda en amplitud. En estos diagramas, los vértices y aristas del árbol están en negro, los vértices no vistos están sombreados y los vértices del margen están en blanco.

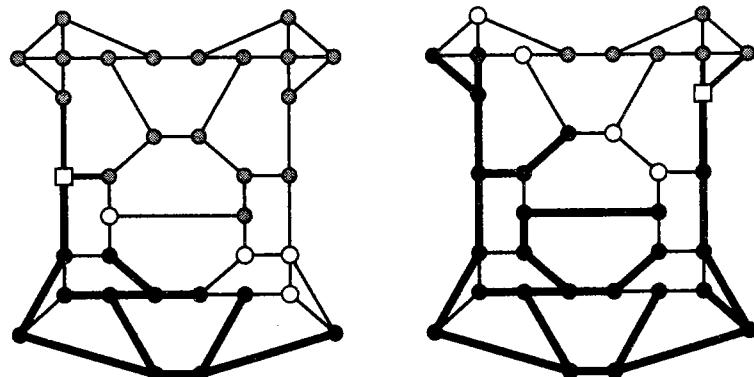


Figura 29.14 Búsqueda en amplitud en un gran grafo.

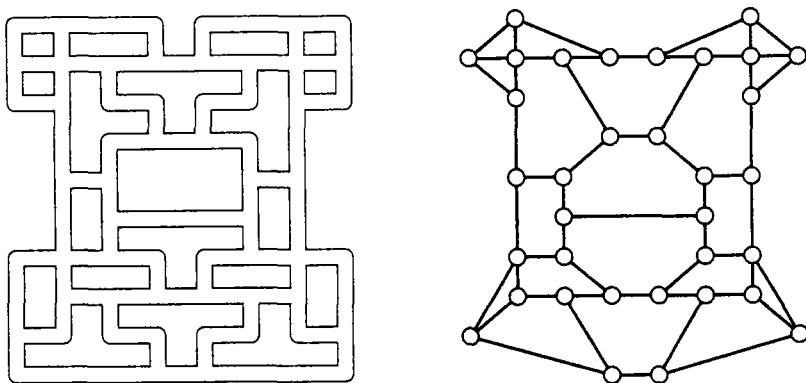


Figura 29.15 Un laberinto y el grafo asociado.

En ambos casos el recorrido comienza en el nodo inferior izquierdo. La búsqueda en profundidad «se sumerge» en el grafo, almacenando en la pila los puntos de los que emergen otros caminos; la búsqueda en amplitud «barre» el grafo, utilizando una cola para memorizar la frontera de los lugares ya visitados. La búsqueda en profundidad «explora» el grafo buscando nuevos vértices lejos del punto de partida, tomando los vértices próximos solamente cuando se encuentre un callejón sin salida; la búsqueda en amplitud cubre completamente la zona cercana al punto de partida, alejándose solamente cuando todos los vecinos han sido vistos. Una vez más, el orden en el que se visitan los nodos depende fuertemente del orden en el que aparecen las aristas en la entrada y de los efectos de esta ordenación en la forma como aparecen los vértices en las listas de adyacencia.

Más allá de estas diferencias operativas, es interesante reflejar las diferencias fundamentales que existen entre las implementaciones de estos métodos. La búsqueda en profundidad se expresa simplemente de forma recursiva (dado que la estructura de datos subyacente es una pila) y la búsqueda en amplitud admite una implementación no recursiva (dado que la estructura de datos subyacente es una cola). En el Capítulo 31 se verá que la estructura de datos subyacente de los algoritmos sobre grafos es realmente una cola de prioridad, lo que implica una abundancia de interesantes propiedades y algoritmos.

Laberintos

Esta forma sistemática de visitar cada vértice y arista de un grafo tiene una historia muy particular: la búsqueda en profundidad fue expuesta formalmente hace

centenares de años como un método para recorrer laberintos. Por ejemplo, en la parte izquierda de la Figura 29.15 se representa un popular laberinto, y a la derecha el grafo construido al colocar un vértice en cada punto en el que existe más de un camino a tomar y al conectar a continuación los vértices de acuerdo con esos caminos. Este laberinto es claramente más complicado que los de los antiguos jardines ingleses, que estaban construidos como caminos rodeados de grandes setos. En estos laberintos todos los muros estaban conectados a otros muros, por lo que damas y caballeros podían pasear por ellos, y los más inteligentes de ellos podían encontrar la salida siguiendo simplemente el muro de su mano derecha (los ratones de laboratorio han aprendido estos trucos). Cuando pueden existir paredes interiores independientes se necesita una estrategia más sofisticada, lo que conduce a una búsqueda en profundidad.

Para desplazarse de un lugar a otro del laberinto por medio de una búsqueda en profundidad se utiliza visitar partiendo del vértice del grafo que corresponde al punto de partida. Cada vez que visitar «sigue» una arista por medio de una llamada recursiva, se marcha a lo largo del camino correspondiente del laberinto. El truco consiste en *retroceder* por el camino que se ha utilizado para llegar a cada vértice una vez que visitar ha terminado con dicho vértice. Así se vuelve al vértice situado justo un nivel por encima en el árbol de búsqueda en profundidad, y se está preparado para seguir por la arista adyacente. (Este proceso reproduce fielmente el recorrido de la búsqueda en profundidad de un grafo.) La búsqueda en profundidad es apropiada para la búsqueda de un elemento del laberinto por una sola persona, dado que el «siguiente lugar a visitar» está siempre próximo; la búsqueda en amplitud es más apropiada para un grupo de personas que buscan el mismo elemento desplazándose en todas las direcciones a la vez.

Perspectivas

En los capítulos siguientes se verá una serie de algoritmos sobre grafos enfocados principalmente a la determinación de las propiedades de la conectividad relativas a los grafos, dirigidos o no. Estos algoritmos son fundamentales para el tratamiento de grafos, pero son solamente una introducción al tema de los algoritmos sobre grafos. Se han desarrollado muchos algoritmos útiles e interesantes que están fuera del alcance de este libro y se han estudiado muchos problemas interesantes para los que no se ha conocido ningún algoritmo eficaz.

Algunos de los algoritmos que se han desarrollado son demasiado complejos para presentarlos aquí. Por ejemplo, es posible determinar de forma eficaz si un grafo puede representarse (o no) en un plano sin que haya ninguna intersección entre sus líneas. Este problema, denominado de *planaridad*, ha debido esperar hasta 1974 para conocer un algoritmo que lo resuelva. Fue R.E. Tarjan quien en ese año desarrolló un ingenioso algoritmo (aunque muy complejo) para resolver el problema en tiempo lineal, utilizando la búsqueda en profundidad.

Algunos problemas sobre grafos que se encuentran de forma natural y son fáciles de formular, parecen de difícil resolución y no existen algoritmos conocidos para resolverlos. Por ejemplo, no se conoce ningún algoritmo eficaz para encontrar el camino de coste mínimo que explora todos los vértices de un grafo ponderado. Este problema, denominado el *problema del vendedor ambulante*, pertenece a una amplia clase de problemas difíciles que se presentarán con más detalle en el Capítulo 45. La mayor parte de los expertos están convencidos de que no existe ningún algoritmo eficaz para estos problemas.

Otros problemas sobre grafos pueden contar con algoritmos eficaces, aunque no se haya encontrado todavía ninguno. Un ejemplo de ellos es el problema del *isomorfismo de grafos* en el que se desea conocer si es posible identificar a dos grafos renombrando simplemente sus vértices. Se conocen algoritmos eficaces para este problema en muchos tipos particulares de grafos, pero el problema general permanece abierto.

En resumen, existe un amplio espectro de problemas y algoritmos para tratar a los grafos. Ciertamente no se puede esperar resolver todos los problemas que se encuentren y, al contrario, algunos problemas que parecen simples todavía causan confusión en los expertos. Pero lo normal es que aparezcan con frecuencia un cierto número de problemas relativamente simples, y, además, los algoritmos sobre grafos que se estudiarán en este libro serán muy útiles en una gran variedad de aplicaciones.

Ejercicios

1. ¿Qué representación de grafo no dirigido es más apropiada para determinar rápidamente si un vértice está aislado (no conectado a ningún otro vértice) o no lo está?
2. Supóngase que se utiliza una búsqueda en profundidad sobre un árbol binario de búsqueda y que la arista derecha se toma antes que la izquierda, al salir de cada nodo. ¿En qué orden se visitarán los nodos?
3. ¿Cuántos bits de almacenamiento se necesitan para representar la matriz de adyacencia de un grafo no dirigido de V nodos y A aristas? ¿Cuántos bits se necesitan para la representación con listas de adyacencia?
4. Dibujar un grafo que no pueda representarse en una hoja de papel sin que dos de sus aristas se crucen.
5. Escribir un programa para suprimir una arista de un grafo representado con listas de adyacencia.
6. Escribir una versión de `listaady` que guarde las listas de adyacencia en una ordenación según los índices de los vértices. Analizar las ventajas de esta estrategia.
7. Dibujar el bosque de búsqueda en profundidad que se obtiene del ejemplo del texto cuando el procedimiento `buscar` explora los vértices en orden inverso (de V a 1) para las dos representaciones.

8. ¿Cuántas veces se debe invocar exactamente a `visitar` en una búsqueda en profundidad en un grafo no dirigido? Determinar lo anterior en función del número de vértices V , del número de aristas A y del número de componentes conexas C .
9. Presentar las listas de adyacencia obtenidas si las aristas del grafo del ejemplo se leen en orden inverso al que se utilizó para hacer la estructura de la Figura 29.4.
10. Presentar el bosque de búsqueda en profundidad para el grafo del ejemplo del texto cuando la rutina recursiva se utiliza en las listas de adyacencia del ejercicio anterior.

Conec**t**ividad

El procedimiento fundamental de búsqueda en profundidad del capítulo anterior encuentra las componentes conexas de un grafo dado; en este capítulo se examinarán los algoritmos relacionados y los problemas que conciernen a otras propiedades de la conectividad de los grafos.

Después de haber visto algunas aplicaciones directas de la búsqueda en profundidad para obtener información de conectividad, se examinará una generalización de la conectividad denominada *biconectividad*, cuyo interés reside en conocer si hay más de un medio de pasar de un vértice de un grafo a otro. Un grafo es *biconexo* si, y sólo si, existen al menos dos caminos diferentes que conecten cada par de vértices. De esta forma, si se suprime un vértice y todas las aristas que inciden en él, el grafo permanece conexo. Si para algunas aplicaciones es importante que un grafo *sea* conexo, es también importante que *permanezca* conexo. La solución a este problema es un algoritmo verdaderamente más complejo que los algoritmos de recorrido del capítulo anterior, aunque se base también en la búsqueda en profundidad.

Una versión particular del problema de la conectividad, que con frecuencia concierne a la situación dinámica en la que las aristas se añaden al grafo una a una, intercalando preguntas sobre si dos vértices determinados pertenecen (o no) a la misma componente conexa. Este problema se ha estudiado profundamente, y en este libro se examinarán con detalle dos algoritmos «clásicos» relacionados con él. Estos métodos no solamente son sencillos y de aplicación general, sino que también muestran la gran dificultad que puede existir al analizar algoritmos simples. El problema se denomina a veces como «unión-pertenencia», una nomenclatura que se deriva de la aplicación de los algoritmos al tratamiento de operaciones simples en conjuntos de elementos.

Componentes conexas

Cualquier método de recorrido de grafos del capítulo anterior puede utilizarse para encontrar las componentes conexas de un grafo, dado que todos se basan en la misma estrategia general de visitar todos los nodos de una componente conexa antes de trasladarse a la siguiente. Una forma sencilla de listar las componentes conexas es modificar uno de los programas de búsqueda recursiva en profundidad para que visitar enumere el vértice que se acaba de visitar (es decir, imprimiendo `nombre(k)` justo antes de acabar), y dando a continuación alguna indicación del comienzo de una nueva componente conexa justo antes de la llamada (no recursiva) a visitar en buscar. Esta técnica produciría la siguiente salida cuando se utiliza la búsqueda en profundidad (buscar y la versión de lista de adyacencia de visitar del Capítulo 29) en el grafo del ejemplo (Figura 29.1):

```
G D E F C B A
I H
K M L J
```

Otras variantes, como la versión de matriz de adyacencia de visitar, la búsqueda en profundidad basada en una pila y la búsqueda en amplitud, pueden calcular las mismas componentes conexas (por supuesto), pero los vértices se imprimirán en un orden diferente.

Es fácil obtener extensiones para hacer más complejo el procesamiento de componentes conexas. Por ejemplo, insertando simplemente `inval[id]=k` después de la instrucción `val[k]=id` se obtiene la «inversa» del array `val`, cuya `i`-ésima entrada es el índice del `i`-ésimo vértice explorado. Los vértices de la misma componente conexa están contiguos en este array; el índice de cada nueva componente conexa está dado por el valor de `i` cada vez que se llama a visitar en buscar. Estos valores pueden almacenarse o utilizarse como delimitadores en `inval` (por ejemplo, la primera entrada de cada componente conexa podría hacerse negativa).

<code>k</code>	1	2	3	4	5	6	7	8	9	10	11	12	13
<code>nombre[k]</code>	A	B	C	D	E	F	G	H	I	J	K	L	M
<code>val[k]</code>	1	7	6	5	3	2	4	8	9	10	11	12	13
<code>inval[k]</code>	-1	6	5	7	4	3	2	-8	9	-10	11	12	13

Figura 30.1 Estructuras de datos para componentes conexas.

La Figura 30.1 muestra los valores tomados por estos arrays del ejemplo si la versión lista de adyacencia de buscar se modificara de esta manera. Normalmente merece la pena utilizar tales técnicas para dividir un grafo en sus componentes conexas y más adelante procesarlo por medio de algoritmos más

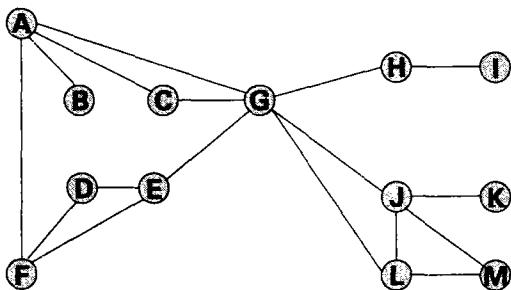


Figura 30.2 Un grafo que no es biconexo.

sofisticados, de forma que se les libere de los detalles de tratar con componentes no conexas.

Biconectividad

A veces es útil diseñar más de una ruta entre puntos de un grafo, aunque sólo sea para identificar posibles fallos en los puntos de conexión (vértices). Así se puede volar desde Providence a Princeton, aunque New York esté cerrado por nieve, sin tener que ir por Philadelphia. Las principales líneas de conexión de un circuito integrado son a menudo biconexas, por lo que el resto del circuito puede continuar funcionando si falla uno de los componentes. Otra aplicación (no muy realista, pero que da una ilustración natural del concepto) es la de imaginar una situación de guerra en la que se obliga al enemigo a bombardear al menos dos estaciones para poder cortar las líneas de ferrocarril.

Un *punto de articulación* en un grafo conexo es un vértice que si se suprime romperá el grafo en dos o más piezas. De un grafo que no tiene puntos de articulación se dice que es *biconexo*. En un grafo biconexo, cada par de vértices están conectados por dos caminos distintos. Un grafo que es no biconexo se divide en *componentes biconexas*, conjuntos de nodos accesibles mutuamente por medio de dos caminos distintos.

En la Figura 30.2 se muestra un grafo que es conexo pero no biconexo. (Este grafo se ha obtenido del correspondiente al del capítulo anterior añadiendo las aristas GC, GH, JG y LG. En los ejemplos se supone que estas cuatro aristas se han añadido al final de los datos de entrada y en el orden anterior, de tal forma, por ejemplo, que las listas de adyacencia sean similares a las de la Figura 29.4 con ocho nuevas entradas correspondientes a las cuatro nuevas aristas.) Los puntos de articulación de este grafo son A (dado que conecta a B con el resto del grafo), H (al conectar a I con el resto del grafo), J (que conecta a K con el

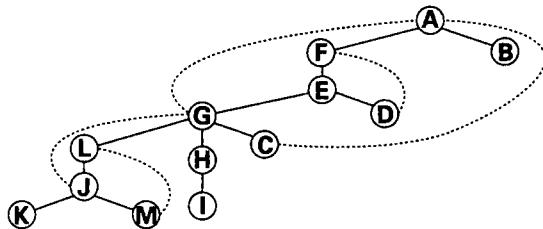


Figura 30.3 Búsqueda en profundidad en la biconectividad.

resto del grafo) y G (dado que el grafo se rompería en tres piezas si se borrara G). Existen seis componentes biconexas: {A C G D E F}, {G J L M}, y los nodos individuales B, H, I y K.

La determinación de los puntos de articulación resulta ser una simple extensión de la búsqueda en profundidad. Para comprobarlo, considérese el árbol de búsqueda en profundidad de este grafo, mostrado en la Figura 30.3. Borrando el nodo E, no se desconecta el grafo, dado que G y D tienen ambos enlaces de puntos por encima de E, que proporcionan caminos alternativos para ir de dichos vértices a F (el padre de E en el árbol). Por el contrario, borrando G se desconecta el grafo porque no existen caminos alternativos para ir de L o H a E (què es el padre de G).

Un vértice x no es un punto de articulación si cada uno de sus hijos y tiene algún nodo descendiente conectado (por medio de un enlace de puntos) a un nodo más alto que x en el árbol, que proporciona una conexión alternativa entre x e y . Esta prueba no es válida al trabajar en la raíz del árbol de búsqueda en profundidad, porque este nodo no tiene «ascendientes en el árbol».

La raíz es un punto de articulación si tiene dos o más hijos, dado que el único camino para conectar a hijos de la raíz pasa por ella misma. Estas pruebas se incorporan fácilmente a la búsqueda en profundidad transformando el procedimiento de exploración de nodos en una función que devuelva el punto más alto del árbol (valor inferior de val) que se haya encontrado durante la búsqueda, de la siguiente forma:

```
int visitar(int k) // BP para encontrar puntos de articulación
{
    struct nodo *t;
    int m, min;
    val[k] = ++id;
    min = id;
    for (t = ady[k]; t != z; t = t->siguiente)
```

```

if (val[t->v] == novisto)
{
    m = visitar(t->v);
    if (m < min) min = m;
    if (m >= val[k]) cout << nombre(k);
}
else if (val[t->v] < min) min = val[t->v];
return min;
}

```

Este procedimiento determina recursivamente el punto más alto de árbol accesible (por medio de un enlace de puntos) desde cualquiera de los descendientes del vértice k , y utiliza esta información para determinar si k es un punto de articulación. Normalmente este cálculo no implica más que comprobar si el valor mínimo accesible desde un hijo está más alto en el árbol, o no lo está. Sin embargo, se necesita una prueba extra para determinar si k es la raíz de un árbol de búsqueda en profundidad (o, de forma equivalente, si se trata de la primera llamada a `visitar` para la componente conexa que contiene a k), ya que se utiliza en ambos casos el mismo programa recursivo. Es conveniente llevar a cabo esta prueba fuera del `visitar` recursivo y así no aparecerá en el código que se acaba de mostrar.

Propiedad 30.1 *Las componentes biconexas de un grafo pueden determinarse en tiempo lineal.*

Aunque el programa anterior no hace más que imprimir los puntos de articulación, es fácil ampliarlo, como en el caso de las componentes conexas, para que efectúe un tratamiento adicional de los puntos de articulación y de las componentes conexas. Al ser un procedimiento de búsqueda en profundidad, el tiempo de ejecución es proporcional a $V + A$. (Un programa similar, basado en una matriz de adyacencia, se ejecutaría en $O(V^2)$ pasos.)■

Además de los tipos de aplicaciones mencionados anteriormente, en las que las biconectividades se han utilizado para mejorar la fiabilidad, pueden ser de una gran ayuda en la descomposición de grafos muy grandes en varias partes más manejables. Es obvio que en muchas aplicaciones puede procesarse un grafo muy grande, haciéndolo componente conexa a componente conexa, pero a veces es más práctico, aunque sea menos evidente, el poder procesar un grafo componente biconexa a componente biconexa.

Algoritmos de unión-pertenencia

En algunas aplicaciones se desea simplemente conocer si un vértice x está o no conectado a un vértice y de un grafo, sin que sea importante el camino que los

conecta de hecho. Este problema se ha estudiado cuidadosamente en los últimos años: los eficaces algoritmos que se han desarrollado son interesantes por sí mismos dado que también pueden utilizarse para el procesamiento de *conjuntos* (colecciones de objetos). Los grafos se corresponden de forma natural con estos conjuntos: los vértices representan a los objetos y las aristas significan «está en el mismo conjunto que....». Así, el grafo ejemplo del capítulo anterior corresponde a los conjuntos {A B C D E F G}, {H I} y {J K L M}. Otro término para definir estos conjuntos es el de *clases de equivalencia*. Cada componente conexa corresponde a una clase de equivalencia diferente. El añadir una arista se corresponde con la combinación de las clases de equivalencia representadas por los vértices a conectar. El interés se centra en la pregunta fundamental «¿es *x* equivalente a *y*?» o «¿está *x* en el mismo conjunto que *y*?». Esto se corresponde claramente con la pregunta fundamental de los grafos «¿está el vértice *x* conectado al vértice *y*?».

Dado un conjunto de aristas, se puede construir una representación por lista de adyacencias que corresponda al grafo y utilizar la búsqueda en profundidad para asignar a cada vértice el índice de su componente conexa, y así preguntas tales como «¿está *x* conectada a *y*?» pueden responderse con dos accesos a arrays y una comparación. La característica suplementaria de los métodos que se considerarán aquí es que son *dinámicos*: pueden aceptar nuevas aristas mezcladas arbitrariamente con preguntas y contestar correctamente a las preguntas utilizando la información recibida. Por correspondencia con el problema de los conjuntos, la adición de una nueva arista se denomina una operación de *unión*, y las preguntas se denominan operaciones de *pertenencia*.

El objetivo es escribir una función que pueda verificar si dos vértices *x* e *y* pertenecen al mismo conjunto (o, en representación de grafos, a la misma componente conexa) y, en caso de que sea así, que pueda unirlos en el mismo conjunto (colocando una arista entre ellos y el grafo). En lugar de construir una lista de adyacencia directa o cualquier otra representación de los grafos, es más eficaz utilizar una estructura interna orientada específicamente a la realización de las operaciones *union* y *pertenencia*. Esta estructura interna es un *bosque de árboles*, uno por cada componente conexa. Se necesita poder encontrar si dos vértices pertenecen al mismo árbol y combinar dos árboles en uno. Por fortuna ambas operaciones pueden implementarse eficazmente.

Para ilustrar el funcionamiento del algoritmo, se examina el bosque que se obtiene cuando las aristas del grafo del ejemplo de la Figura 30.1 se procesan en el orden AG AB AC LM JM JL JK ED FD HI FE AF GE GC GH JG LG. En la Figura 30.4 se muestran los siete primeros pasos. Inicialmente, todos los nodos están en árboles separados. A continuación la arista AG provoca la creación de un árbol dos-nodos de raíz A. (La elección es arbitraria —igualmente se podría haber tomado como raíz a G—.) Las aristas AB y AC añaden a B y C al árbol de la misma forma. A continuación las aristas LM, JM, JL y JK construyen un árbol que contiene a J, K, L y M, que tiene una estructura ligeramente diferente (es de destacar que JL no contribuye con nada, dado que LM y JM colocan a L y J en el mismo componente).

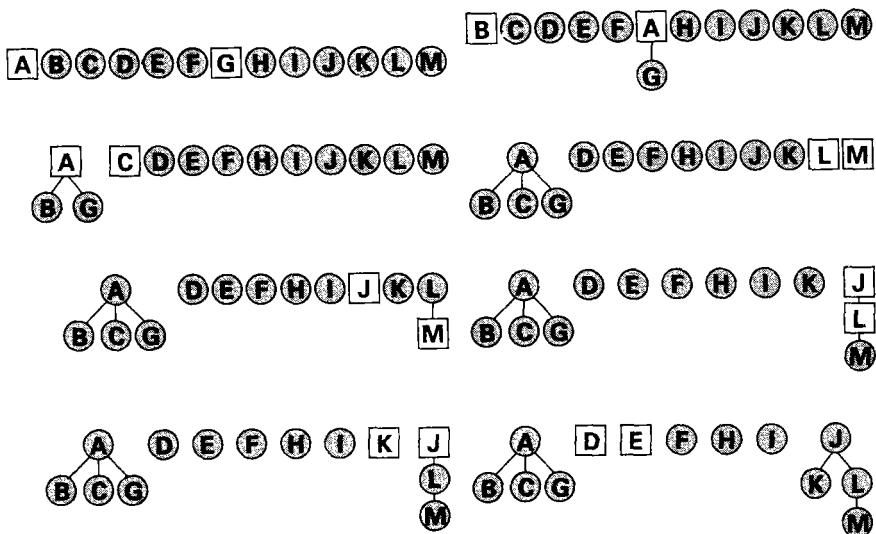


Figura 30.4 Etapas iniciales de unión-pertenencia.

La Figura 30.5 muestra el fin del proceso. Las aristas ED, FD y HI generan dos árboles más, quedando un bosque con cuatro árboles. Este bosque indica que las aristas procesadas hasta este momento describen un grafo con cuatro componentes conexas, o, de forma equivalente, que las operaciones de unión de conjuntos efectuadas hasta el momento conducen a cuatro conjuntos {A B C G}, {J K L M}, {D E F} y {H I}. Ahora la arista FE no contribuye con nada a la estructura, dado que F y E están en la misma componente, pero la arista AF combina los dos primeros árboles; a continuación GE y GC tampoco contribuyen con nada, pero GH y JG reúnen todo el bosque en un solo árbol.

Se debe enfatizar que, a diferencia de los árboles de búsqueda en profundidad, la única relación entre estos árboles de unión-pertenencia y el grafo asociado con las aristas dadas es que aquéllos dividen de forma análoga a los vértices en conjuntos. Por ejemplo, no hay correspondencia entre los caminos que conectan los nodos de los árboles y los que conectan los nodos de los grafos. ¿Qué estructura de datos debe elegirse para mantener estos bosques? Aquí sólo se recorren los árboles *hacia arriba*, nunca hacia abajo, por lo que es apropiada la representación de «enlace padre» (ver el Capítulo 4). Específicamente, se mantiene un array *padre* que contiene, para cada vértice, el índice de su padre (0 si es la raíz de algún árbol), como se especifica en la siguiente declaración clase:

```
clase Eq
```

```
{
```

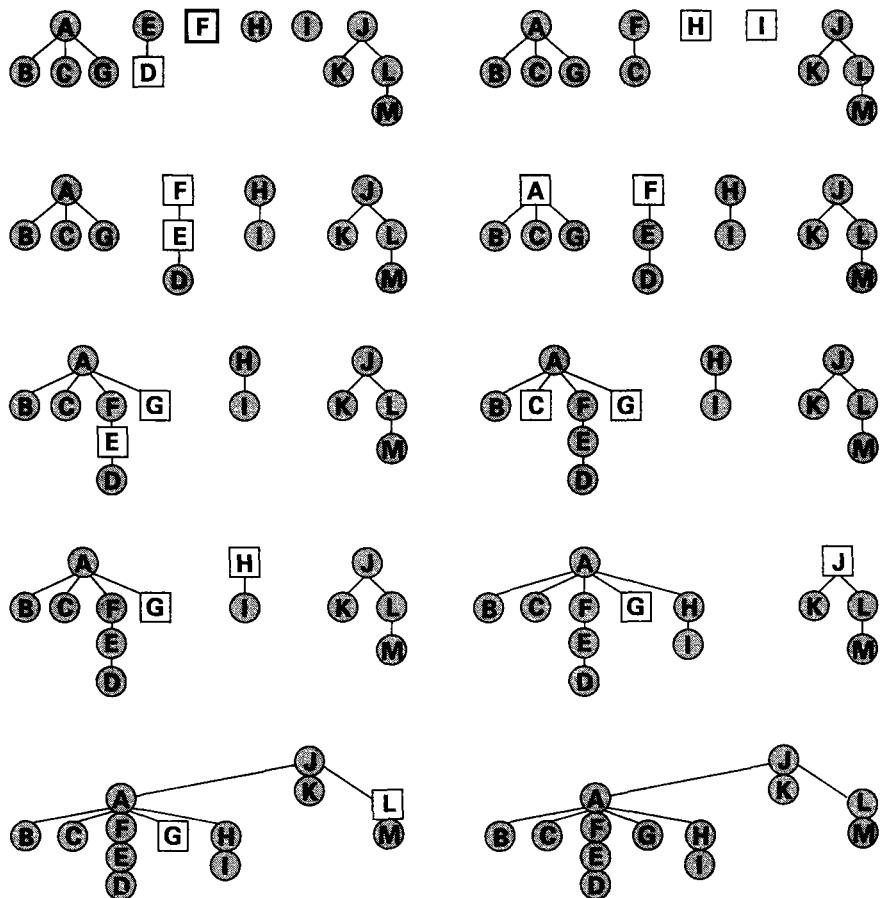


Figura 30.5 Final de unión-pertenencia.

```

private:
    int *padre;
public:
    EQ(int talla);
    int pertenencia(int x, int y, int union);
};
```

El constructor asigna espacio para el array `padre` e inicialmente coloca todos sus valores a cero (aquí se omite el código). Se utiliza un sencillo procedimiento `pertenencia` para implementar las dos operaciones, *unión* y *pertenencia*. Si el tercer argumento es 0 se tiene solamente una *pertenencia*; si es distinto de cero, se tiene una *unión*. Para encontrar el padre de un vértice j , simplemente se establece $j = \text{padre}[j]$ y para encontrar la raíz del árbol que contiene a j se re-

pite esta operación hasta obtener 0. Esto conduce a una implementación íntegra del procedimiento pertenencia:

```
int EQ::pertenencia(int x, int y, int unión)
{
    int i = x, j = y;
    while (padre[i] > 0) i = padre[i];
    while (padre[j] > 0) j = padre[j];
    if (union && (i != j)) padre[j] = i;
    return (i != j);
}
```

La función pertenencia devuelve 0 si los dos vértices dados son de la misma componente. Si no es así y el indicador union está activado, se colocarán dichos vértices en la misma componente. El método es simple: se utiliza un array padre para obtener la raíz del árbol que contiene a cada vértice, y a continuación se comprueba que las dos raíces son idénticas. Para fusionar el árbol de raíz j con el de raíz i, se pone simplemente padre[j] = i.

La Figura 30.6 muestra el contenido de la estructura de datos durante el proceso. Como de costumbre, se supone que las funciones índice y nombre permiten la traducción de los nombres de los vértices en enteros entre 1 y V: cada entrada de la tabla es el nombre de la correspondiente entrada del array padre. Por ejemplo, después de declarar una instancia de la clase EQ (con la declaración EQ eq(max)), se comprobará si un vértice denominado x está en la misma componente que un vértice denominado y (sin la introducción de una arista entre ellos) llamando a la función eq.pertenencia(indice(x), índice(y), 0).

Esta clase está codificada de forma que puede usarse en cualquier aplicación en la que las clases de equivalencia tengan un papel importante, no sólo en la conectividad de grafos. El único requisito es que los elementos del conjunto tengan nombres enteros que sirvan como índices (desde 1 hasta V). Como ya se dijo, se pueden usar las implementaciones de diccionario anteriores para que así sea.

El algoritmo descrito anteriormente tiene un mal comportamiento en el peor caso porque los árboles construidos pueden estar degenerados. Por ejemplo, si se toman las aristas en el orden AB BC CD DE EF FG GH HI IJ ... YZ se obtiene una gran cadena en la que Z apunta a Y, Y apunta a X, etc. Para construir este tipo de estructura se necesita un tiempo que es proporcional a V^2 , y para una comprobación de equivalencia, un tiempo proporcional a V .

Para resolver este problema se han propuesto diversas técnicas. Un método natural, que posiblemente ya se le haya ocurrido al lector, es intentar hacer lo más «razonable» cuando se mezclan dos árboles, en lugar de poner arbitrariamente $padre[j] = i$. Cuando se va a mezclar un árbol de raíz i con otro de raíz j, uno de los nodos debe permanecer como raíz y el otro (y todos sus des-

	A	B	C	D	E	F	G	H	I	J	K	L	M
AG							A						
AB	A						A						
AC	A	A					A						
LM	A	A					A				L		
JM	A	A					A			J	L		
JL	A	A					A			J	L		
JK	A	A					A			J	J	L	
ED	A	A	E			A			J	J	L		
FD	A	A	E	F		A			J	J	L		
HI	A	A	E	F		A	H		J	J	L		
FE	A	A	E	F		A	H		J	J	L		
AF	A	A	E	F	A	A	H		J	J	L		
GE	A	A	E	F	A	A	H		J	J	L		
GC	A	A	E	F	A	A	H		J	J	L		
GH	A	A	E	F	A	A	A	H		J	J	L	
JG	J	A	A	E	F	A	A	A	H	J	J	L	
LG	J	A	A	E	F	A	A	A	H	J	J	L	

Figura 30.6 Estructura de datos unión-pertenencia.

cendientes) debe bajar un nivel en el árbol. Para hacer mínima la distancia entre la raíz y la mayor parte de los nodos, parece lógico elegir como raíz el nodo que tenga el mayor número de descendientes. Esta idea, denominada *equilibrado de peso*, se implementa fácilmente manteniendo el tamaño de cada árbol (número de descendientes de la raíz) en la entrada del array padre, para todos los nodos raíz, codificando como números negativos a todos los nodos raíz que puedan detectarse ascendiendo por el árbol en pertenencia.

Lo ideal sería arreglárselas para que todos los nodos apuntaran directamente a la raíz de su árbol. Sin embargo, independientemente de la estrategia a utilizar, para lograr este ideal habría que examinar al menos todos los nodos de uno de los dos árboles a fusionar, y esto podría representar una cantidad muy grande comparada con los relativamente pocos nodos situados en el camino hacia la raíz que suele examinar pertenencia. Pero se puede hacer una aproximación al ideal haciendo que todos los nodos que se examinan apunten hacia la raíz! A primera vista esta operación parece ser muy drástica, pero es muy fácil de

hacer, y no hay nada sagrado en lo que respecta a la estructura de estos árboles: si es factible modificarlos para hacer que el algoritmo sea más eficaz, debe hacerse. Este método, denominado *compresión de caminos*, es fácil de implementar haciendo otra pasada a través de cada árbol, después de haber encontrado la raíz, y fijando la entrada padre de cada vértice que se ha encontrado a lo largo del camino que apunta a la raíz.

La combinación de los métodos de equilibrado de peso y compresión de caminos asegura que los algoritmos se ejecuten más rápidamente. La siguiente implementación muestra que el código extra es un precio a pagar muy pequeño para prevenirse de los casos degenerados.

```
int EQ::pertenencia(int x, int y, int union)
{
    int i = x, j = y;
    while (padre[i] > 0) i = padre[i];
    while (padre[j] > 0) j = padre[j];
    while (padre[x] > 0)
        { t = x; x = padre[x]; padre[t] = i; }
    while (padre[y] > 0)
        { t = y; y = padre[y]; padre[t] = j; }
    if (unión && (i != j))
        if (padre[j] < padre[i])
            { padre[j] += padre[i] - 1; padre[i] = j; }
        else
            { padre[i] += padre[j] - 1; padre[j] = i; }
    return (i != j);
}
```

La Figura 30.7 muestra los primeros ocho pasos de la aplicación del método a los datos del ejemplo y la Figura 30.8 muestra el final del proceso. La longitud media de camino del árbol resultante es $31/13 \approx 2,38$, a comparar con el valor de $38/13 \approx 2,92$ de la Figura 30.5. Para las cinco primeras aristas el bosque resultante es el mismo que el de la Figura 30.4; sin embargo, las tres últimas aristas producen un árbol «plano» que contiene a J, K, L y M a causa de la regla del equilibrado de peso. Los bosques de este ejemplo son tan planos que todos los vértices implicados en las operaciones de *unión* están en la raíz o justo debajo —no se utiliza la compresión de caminos (ya que podría hacer que los árboles fueran todavía más planos)—. Por ejemplo, si la última *unión* fue FJ y no GL, entonces al final F acabaría siendo un hijo de A.

La Figura 30.9 proporciona el contenido del array *padre* según se va construyendo el bosque. Para mayor claridad de la tabla, cada entrada positiva *i* se ha reemplazado por la *i*-ésima letra del alfabeto (el nombre del parente) y a cada entrada negativa se ha aplicado el complemento para obtener un entero positivo (el peso del árbol).

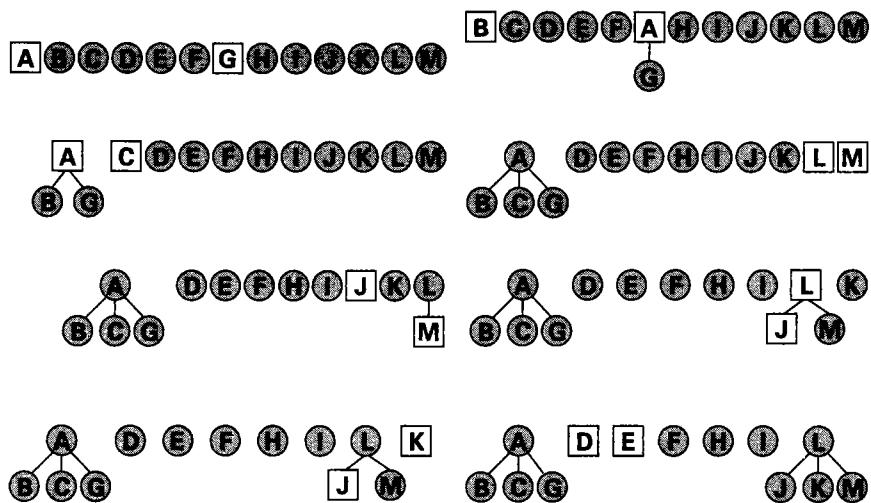


Figura 30.7 Primeros pasos de unión-pertenencia (equilibrado, con compresión de caminos).

Se han desarrollado otras muchas técnicas para evitar las estructuras degeneradas. Por ejemplo, la compresión de caminos tiene el inconveniente de necesitar una segunda pasada a través del árbol. Otra técnica, denominada *división*, hace que cada nodo apunte a su abuelo en el árbol. Otra más, la *escisión*, es como la anterior, pero se aplica solamente a uno de cada dos nodos del camino de búsqueda. Cualquiera de estas técnicas puede utilizarse conjuntamente con el equilibrado de peso o con el *equilibrado de altura*, que es similar pero utiliza la altura del árbol para decidir cómo fusionar los árboles.

¿Qué método se debe elegir entre todos éstos? y ¿hasta qué punto son «planos» los árboles generados? El análisis de este problema es muy difícil dado que el rendimiento depende no solamente de los parámetros V y A , sino también del número de operaciones de *pertenencia* y, lo que es peor, del orden en que se efectúan las operaciones de *unión* y *pertenencia*. A diferencia de la ordenación, en la que los archivos reales que aparecen en la práctica son a menudo casi «aleatorios», es difícil ver los modelos de grafos y consultas que pueden aparecer en la práctica. Por esta razón los algoritmos que tienen un buen comportamiento en el peor caso se prefieren a los de unión-pertenencia (y a otros algoritmos sobre grafos), aunque esto puede ser un enfoque superconservador.

Incluso si sólo se considera el peor caso, el análisis de los algoritmos de unión-pertenencia es extremadamente complejo e intrincado. Esto puede deducirse de la naturaleza de los resultados, que sin embargo dan una idea clara del comportamiento de los algoritmos en una situación práctica.

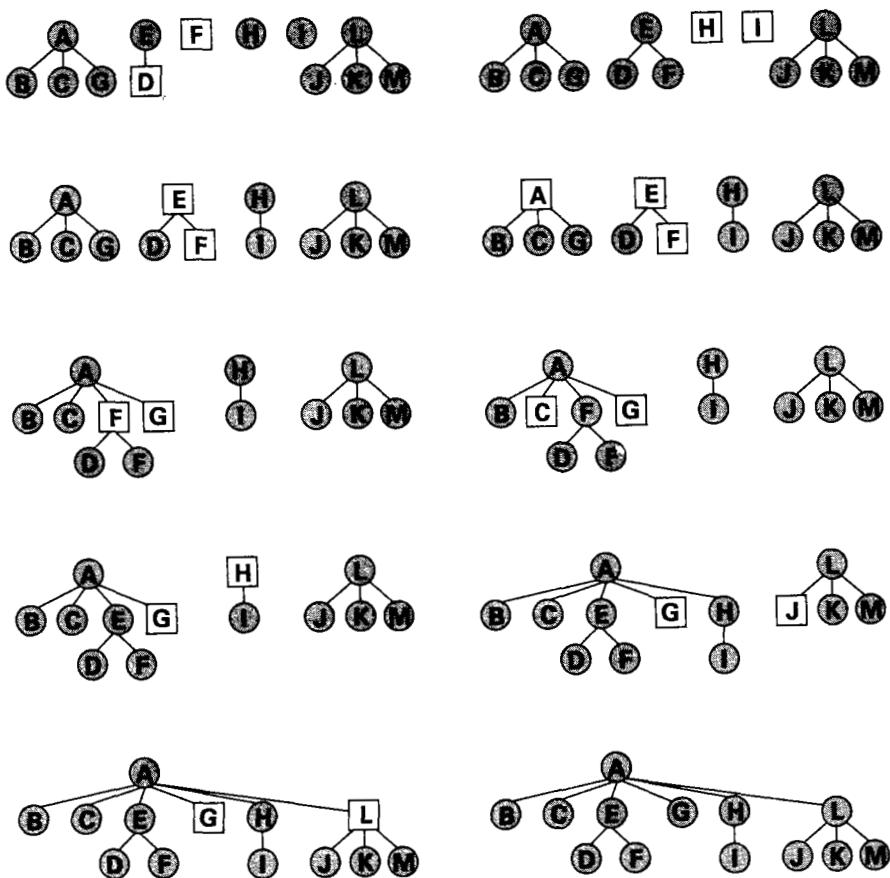


Figura 30.8 Final de unión-pertenencia (equilibrado, con compresión de caminos).

Propiedad 30.2 Si se utiliza un equilibrado de peso o de altura conjuntamente con la compresión, división o escisión, el número total de operaciones necesarias para la construcción de una estructura utilizando A aristas es casi (pero no exactamente) lineal.

Con más precisión, el número de operaciones necesarias es proporcional a $Aa(A)$, donde $a(A)$ es una función que crece tan lentamente que $a(A) < 4$, a menos que A sea inferior a un valor tan grande que cuando se tome $\lg A$, a continuación se tome el \lg del resultado, y así otra y otra vez, repitiéndolo 16 veces, todavía se obtiene un número ¡mayor que 1! Este número es increíblemente grande: es bastante seguro suponer que la media de tiempo de ejecución de cada operación de *unión* y *pertenencia* es constante. Este resultado se debe a R. E. Tarjan, quien además demostró que *ningún* algoritmo para este problema (dentro de una clase

	A	B	C	D	E	F	G	H	I	J	K	L	M
AG	1						A						
AB	2	A					A						
AC	3	A	A				A						
LM	3	A	A				A			1	L		
JM	3	A	A				A		L	2	L		
JL	3	A	A				A		L	2	L		
JK	3	A	A				A		L	L	3	L	
ED	3	A	A	E	1	A			L	L	3	L	
FD	3	A	A	E	2	E	A		L	L	3	L	
HI	3	A	A	E	2	E	A	1	H	L	L	3	L
FE	3	A	A	E	2	E	A	1	H	L	L	3	L
AF	6	A	A	E	A	E	A	1	H	L	L	3	L
GE	6	A	A	E	A	E	A	1	H	L	L	3	L
GC	6	A	A	E	A	E	A	1	H	L	L	3	L
GH	8	A	A	E	A	E	A	A	H	L	L	3	L
JG	12	A	A	E	A	E	A	A	H	L	L	A	L
LG	12	A	A	E	A	E	A	A	H	L	L	A	L

Figura 30.9 Estructura de datos de unión-pertenencia (equilibrada, con compresión de caminos).

general de ellos) puede hacerlo mejor que $Aa(A)$, por lo que esta función es intrínseca al problema.■

Una importante aplicación práctica de los algoritmos de unión-pertenencia es determinar si un grafo de V vértices y A aristas es conexo en tiempo proporcional a V (y casi en tiempo lineal). Ésta es una ventaja sobre la búsqueda en profundidad en algunos casos: aquí no se necesita almacenar las aristas. Por ello la conectividad de un grafo con miles de vértices y millones de aristas puede determinarse por medio de una pasada rápida a través de las aristas.

Ejercicios

1. Encontrar los puntos de articulación y las componentes biconexas del grafo que se obtiene al suprimir GJ y añadir IK al grafo de ejemplo.
2. Dibujar el árbol de búsqueda en profundidad del grafo descrito en el ejercicio 1.
3. ¿Cuál es el número mínimo de aristas que se necesitan para construir un grafo biconexo con V vértices?
4. Escribir un programa para imprimir las componentes biconexas de un grafo.
5. Dibujar el bosque de unión-pertenencia construido para el ejemplo del texto, pero suponiendo que pertenencia se cambia para poner $a[i]=j$ en lugar de $a[j]=i$.
6. Resolver el ejercicio anterior, suponiendo además que se utiliza la compresión de caminos.
7. Dibujar el bosque de unión-pertenencia construido por las aristas AB BC CD DE EF ... YZ, suponiendo en primer lugar que se utiliza el método de equilibrado de peso sin compresión de caminos y después este último sin equilibrado de peso.
8. Resolver el ejercicio anterior suponiendo que se utilizan a la vez los dos métodos de compresión de caminos y de equilibrado de peso.
9. Implementar las variantes de unión-pertenencia descritas en el texto y determinar empíricamente la comparación de sus rendimientos para 1.000 operaciones *unión* con argumentos aleatorios enteros comprendidos entre 1 y 100.
10. Escribir un programa para generar un grafo aleatorio conexo con V vértices por generación de pares de enteros aleatorios entre 1 y V . Estimar en función de V qué número de aristas se necesitan para producir un grafo conexo.

Grafos ponderados

Con frecuencia se desea modelar problemas prácticos utilizando grafos en los que se asocia a las aristas unos *pesos* o *costes*. En un mapa de líneas aéreas, en el que las aristas representan rutas de vuelo, los pesos pueden representar distancias o tarifas. En un circuito eléctrico, en el que las aristas representan las conexiones, los pesos naturales a utilizar son la longitud o el precio de los cables. En el diagrama de un proyecto los pesos pueden representar el tiempo o el coste de realización de las tareas o bien el retraso en llevarlas a cabo.

Estas situaciones hacen aparecer de forma natural cuestiones como el minimizar costes. En este capítulo se examinarán detalladamente los algoritmos de dos de estos problemas: «encontrar la forma de conectar todos los puntos al menor coste» y «encontrar el camino de menor coste entre dos puntos dados». El primero, que evidentemente se utiliza para los grafos que representan a cosas similares a circuitos eléctricos, se denomina el problema del *árbol de expansión mínimo*; el segundo, que evidentemente se utiliza para los grafos que representan cosas parecidas a los mapas de líneas aéreas, se denomina el problema del *camino más corto*. Estos problemas representan a toda una variedad de los que se suelen encontrar en los grafos ponderados.

Los algoritmos de este capítulo implican búsquedas a través de los grafos y a veces se hacen pensando intuitivamente en los pesos como si fueran distancias: se habla de «el vértice más próximo a x », etc. De hecho, esta predisposición forma parte de la propia nomenclatura del problema del camino más corto. A pesar de ello, es importante recordar que los pesos no son necesariamente proporcionales a la distancia, y podrían representar tiempos o costes o alguna cosa completamente diferente. Cuando los pesos *realmente* representen las distancias, puede que sean más apropiados otros tipos de algoritmos. Este asunto se presentará con mayor detalle al final del capítulo.

La Figura 31.1 muestra un ejemplo de grafo no dirigido ponderado. La forma de representar a los grafos ponderados es obvia: en la representación por matriz de adyacencia, la matriz puede contener pesos de aristas en lugar de valores booleanos y en la representación por estructuras de adyacencia se puede añadir

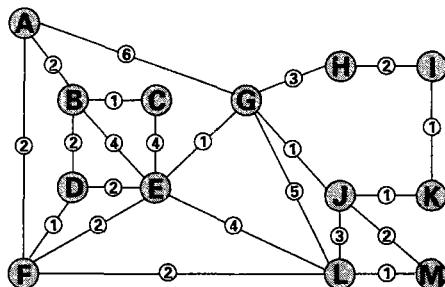


Figura 31.1 Un grafo no dirigido ponderado.

un campo a cada elemento de la lista (que representa una arista), a manera de peso. Se supone que todos los pesos son positivos. Algunos algoritmos se pueden adaptar para manipular pesos negativos, pero se hacen significativamente más complejos. En otros casos, los pesos negativos cambian la naturaleza del problema de forma esencial y se necesita recurrir a algoritmos mucho más sofisticados que los que se consideran aquí. Un ejemplo del tipo de dificultades que pueden encontrarse aparece si se considera la situación en la que la suma de los pesos de las aristas de un ciclo es negativa: un camino infinitamente corto podría engendrarse simplemente dando vueltas alrededor del ciclo.

Se han desarrollado varios algoritmos «clásicos» para resolver los problemas del árbol de expansión mínimo y del camino más corto. Estos métodos se encuentran entre los más célebres y los más utilizados de los algoritmos de este libro. Como se vio anteriormente al estudiar antiguos algoritmos, los métodos clásicos proporcionan un enfoque general, pero las modernas estructuras de datos ayudan a proporcionar implementaciones compactas y eficaces. En este capítulo se verá cómo utilizar colas de prioridad en la generalización de los métodos de recorrido de grafos del Capítulo 29 para resolver ambos problemas de forma eficaz en los grafos dispersos; más adelante se verá la relación entre este método y los clásicos de los grafos densos, y por último se verá un método para resolver el problema del árbol de expansión mínimo que utiliza un enfoque totalmente diferente.

Árbol de expansión mínimo

Un *árbol de expansión mínimo* de un grafo ponderado es una colección de aristas que conectan todos los vértices y en el que la suma de los pesos de las aristas es al menos inferior a la suma de los pesos de cualquier otra colección de aristas que conecten todos los vértices. El árbol de expansión mínimo no es necesaria-

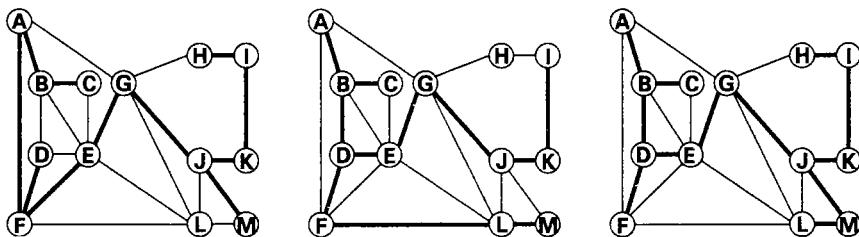


Figura 31.2 Árboles de expansión mínimos.

mente único: la Figura 31.2 muestra los árboles de expansión mínimos del grafo del ejemplo. Es fácil demostrar que la «colección de aristas» de la anterior definición deben formar un árbol de expansión: si existe algún ciclo, se puede suprimir alguna arista del mismo para dar una colección de aristas que todavía conectan a los vértices, pero con un peso inferior.

Se vio en el Capítulo 29 que muchos de los procedimientos de recorrido de grafos construyen un árbol de expansión de dichos grafos. ¿Qué podría hacerse para que, en un grafo ponderado, el árbol construido sea el de peso total más bajo? Existen varias formas de hacerlo, todas basadas en la siguiente propiedad general de los árboles de expansión mínimos.

Propiedad 31.1 *Dada una división cualquiera de los vértices de un grafo en dos conjuntos, el árbol de expansión mínimo contiene la menor de las aristas que conectan un vértice de uno de los conjuntos con un vértice del otro.*

Por ejemplo, dividir los vértices del grafo del ejemplo en dos conjuntos $\{A\ B\ C\ D\}$ y $\{E\ F\ G\ H\ I\ J\ K\ L\ M\}$, implica que DF debe estar en el árbol de expansión mínimo. Esta propiedad es fácil de demostrar por reducción al absurdo. Se denomina a a la menor de las aristas que conectan a los dos conjuntos y se supone que a no está en el árbol de expansión mínimo. A continuación se considera el grafo formado al añadir a al árbol de expansión mínimo. Este grafo tiene un ciclo, que debe contener alguna otra arista diferente de a que conecte a los dos conjuntos. Suprimiendo esta arista y añadiendo a , se obtiene un árbol de expansión de menor peso, lo que contradice el supuesto de que a no se encuentra en el árbol de expansión mínimo.■

Por lo tanto se puede construir el árbol de expansión mínimo comenzando en cualquier vértice y tomando siempre el vértice «más próximo» de todos los que ya se han elegido. En otras palabras, se busca la arista de menor peso entre todas las que conectan vértices que ya están en el árbol con vértices que no lo están todavía, y después se añade al árbol la arista y el vértice a los que conduce la anterior. (En caso de empate, se puede elegir cualquiera de las aristas «empatadas».) La propiedad 31.1 garantiza que cada arista añadida forma parte del árbol de expansión mínimo.

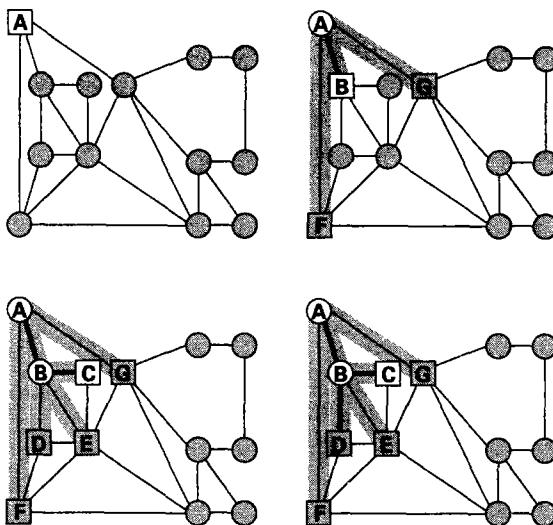


Figura 31.3 Pasos iniciales de la construcción de un árbol de expansión mínimo.

La Figura 31.3 muestra los cuatro primeros pasos cuando esta estrategia se utiliza en el grafo ejemplo, comenzando con el nodo A. El vértice más «próximo» a A (conectado con una arista de peso mínimo) es B, por lo que AB es el árbol de expansión mínimo. De todas las aristas adyacentes a AB, la BC es la de menor peso, así que se añade al árbol, y el vértice C es el siguiente a explorar. Entonces, el vértice más próximo a A, B o C es ahora D, por lo que BD se añade al árbol. La continuación de este proceso se muestra en la Figura 31.5, después de presentar la implementación.

¿Cómo implementar realmente esta estrategia? Hasta ahora el lector habrá reconocido seguramente la estructura básica de: vértices del árbol, del margen y no vistos que caracterizan a las estrategias de búsqueda en profundidad y en anchura del Capítulo 29. Resulta que el mismo método sirve, utilizando una *cola de prioridad* (en lugar de una pila o de una cola), para almacenar los vértices del margen.

Búsqueda en primera prioridad

Recuérdese del Capítulo 29 que la búsqueda en grafos puede describirse en términos de la división de los vértices en tres conjuntos: vértices del *árbol*, cuyas aristas ya han sido examinadas, vértices *del margen*, que están en la estructura de datos esperando su tratamiento, y vértices *no vistos*, a los que todavía no se ha llegado. El método fundamental de búsqueda en grafos que se utiliza aquí está basado en el paso «mover un vértice (denominado x desde el margen hacia

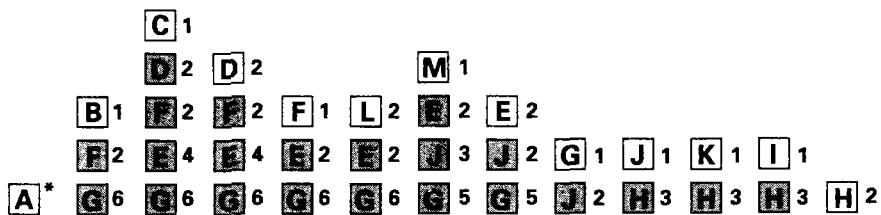


Figura 31.4 Contenido de la cola de prioridad durante la construcción del árbol de expansión mínimo.

el árbol, y colocar después en el margen a cualquiera de los vértices no vistos adyacentes a x . Se utiliza el término de *búsqueda en primera prioridad* para referirse a la estrategia general de la utilización de una cola de prioridad para decidir qué vértice retirar del margen. Esto permite una gran flexibilidad. Como se verá, varios de los algoritmos clásicos (incluyendo las búsquedas en anchura y profundidad) se diferencian solamente en la elección de la prioridad.

Para la construcción del árbol de expansión mínimo, la prioridad de cada vértice del margen debe ser igual a la longitud de la arista más pequeña que lo conecta al árbol. La Figura 31.4 muestra el contenido de una cola de prioridad durante el proceso de construcción presentado en las Figuras 31.3 y 31.5. Para mayor claridad, los elementos de la cola se muestran en orden creciente. Esta implementación por «lista ordenada» de las colas de prioridad podría ser apropiada para grafos pequeños, pero para los grandes deben utilizarse montículos para asegurar que todas las operaciones se puedan finalizar en $O(\log N)$ pasos (véase el Capítulo 11).

En primer lugar, se consideran grafos dispersos con representación por lista de adyacencia. Como se ha mencionado anteriormente, se añade un campo de peso w al registro de aristas (modificando el código de entrada para poder leer pesos). Entonces, si se utiliza una cola de prioridad para el margen, se tiene la siguiente implementación:

```

CP cp(maxV);
visitar(int k) // BPP, listas de adyacencia
{
    struct nodo *t;
    if (cp.actualizar(k, -novisto) != 0) padre[k] = 0;
    while (!cp.vacio())
    {
        id++; k = cp.suprimir(); val[k] = -val[k];
        if (val[k] == -novisto) val[k] = 0;
        for (t = ady[k]; t != z; t = t->siguiente)
            if (val[t->v] < 0)

```

```

        if (cp.actualizar(t->v, prioridad))
        {
            val[t->v] = -(prioridad);
            padre[t->v] = k;
        }
    }
}

```

Para calcular el árbol de expansión mínimo, hay que reemplazar las dos ocurrencias de `prioridad` por `t->w`. Se utiliza la clase diccionario cola de prioridad descrita en el Capítulo 11: la función `actualizar` es una primitiva que se implementa fácilmente y cuyo objetivo es asegurar que el vértice pasado como parámetro aparezca en la cola al menos con la prioridad dada: si el vértice no está en la cola, se aplica una `inserción`, y si el vértice está, pero tiene un gran valor de prioridad, entonces se utiliza `cambiar` para cambiar la prioridad. Si se ha hecho cualquier cambio (o inserción o cambio de prioridad), la función `actualizar` devuelve un valor distinto de cero. Esto permite que el programa anterior mantenga actualizados los arrays `val` y `padre`. El array `val` podría tener él mismo realmente la cola de prioridad de una forma «*indirecta*»; en el programa anterior se han separado las operaciones sobre la cola con prioridad, para una mayor claridad.

A parte del cambio de la estructura de datos por una cola de prioridad, este programa es prácticamente el mismo que se ha utilizado en las búsquedas en anchura y profundidad, con dos excepciones. Primero, se necesita una acción suplementaria cuando se encuentra una arista que ya está en el margen: en las búsquedas en anchura y profundidad se ignoran tales aristas, pero en el programa anterior se necesita comprobar si la nueva arista rebaja la prioridad. Se garantiza así, como es de desear, que siempre se explore el siguiente vértice del margen que es el más próximo al árbol. Segundo, este programa sigue explícitamente la pista del árbol actualizando el array `padre` que almacena al padre de cada nodo en el árbol de búsqueda en primera prioridad (el nombre del nodo que ha provocado su desplazamiento desde el margen hasta el árbol). También, para cada nodo `k` del árbol, `val[k]` es el peso de la arista entre `k` y `padre[k]`. Los vértices del margen se marcan como antes con valores negativos en `val`; el centinela `novisto` toma valores negativos grandes (la razón de esto se hará evidente más adelante).

La Figura 31.5 muestra el final de la construcción del árbol de expansión mínimo del ejemplo. Como es habitual, los vértices del margen se dibujan como cuadrados, los vértices del árbol como círculos y los vértices no vistos como círculos sin etiqueta. Las aristas de los árboles se representan por medio de líneas gruesas en negro, y la arista más pequeña que conecta a cada vértice del margen con el árbol está sombreada. Se anima al lector a seguir la construcción del árbol utilizando las Figuras 31.4 y 31.5. En particular hay que destacar cómo se añade al árbol el vértice `G` después de haber estado en el margen durante varias

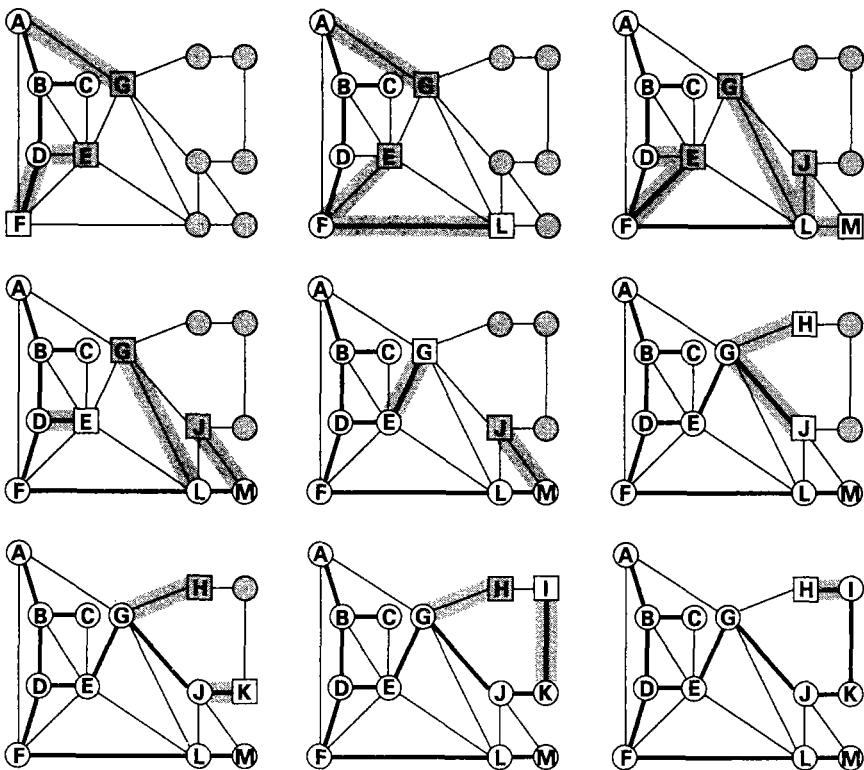


Figura 31.5 Final de la construcción del árbol de expansión mínimo.

etapas. Inicialmente, la distancia desde G al árbol es 6 (por causa de la arista GA). Después de añadir L al árbol, GL disminuye esta distancia hasta 5, y a continuación, después de añadir E, la distancia finalmente baja hasta 1 y G se añade al árbol *antes* de J. La Figura 31.6 muestra la construcción de un árbol de expansión mínimo para el gran grafo «laberinto» anterior, ponderado por la longitud de sus aristas.

Propiedad 31.2 *La búsqueda en primera prioridad en grafos dispersos construye el árbol de expansión mínimo en $O((A+V)\log V)$ etapas.*

Se aplica la propiedad 31.1: los dos conjuntos de nodos en cuestión son los nodos explorados y los sin explorar. En cada etapa se elige la arista más pequeña que conecta un nodo explorado con un nodo del margen (no existen aristas que conecten nodos explorados con nodos no vistos). Así, por la propiedad 31.1, cada arista que se elige está en el árbol de expansión mínimo. La cola de prioridad contiene solamente vértices; si se implementa como un montículo (véase el Ca-

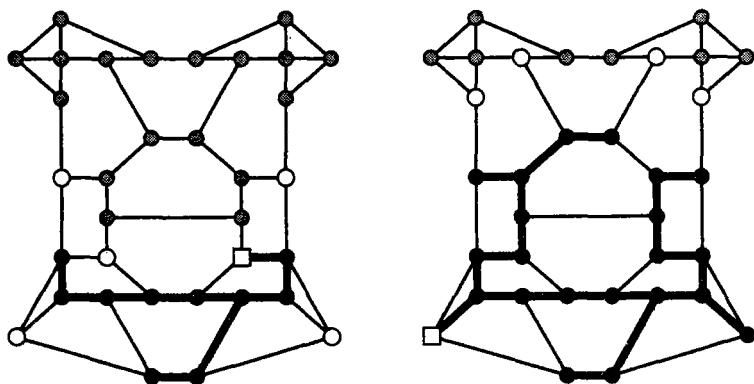


Figura 31.6 Construcción de un gran árbol de expansión mínimo.

pítulo 11), entonces cada operación necesita $O(\log V)$ pasos. Cada vértice conduce a una inserción y cada arista a una operación de cambio de prioridad.■

Se verá posteriormente que este método también resuelve el problema del camino más corto, si se hace una elección apropiada de la prioridad. También se verá que una implementación diferente de la cola de prioridad puede dar un algoritmo en V^2 , que es apropiado para los grafos densos. Esto es equivalente a un antiguo algoritmo que data al menos de 1957: para el árbol de expansión mínimo se atribuye normalmente a R. Prim, y para el camino más corto suele atribuirse a E. Dijkstra. Por coherencia, aquí se hará referencia a estas soluciones (para los grafos densos) como el «algoritmo de Prim» y el «algoritmo de Dijkstra» respectivamente, y se hará referencia al método anterior (para grafos dispersos) como la «solución de la búsqueda en primera prioridad».

La búsqueda en primera prioridad es una buena generalización de las búsquedas en anchura y en profundidad, porque estos métodos pueden deducirse por medio de una adecuada elección de la prioridad. Como id aumenta desde 1 hasta V durante la ejecución del algoritmo, esto se puede utilizar para asignar prioridades únicas a los vértices. Si se cambian las dos ocurrencias de prioridad en `listasbpp` por $V - \text{id}$, se obtiene una búsqueda en profundidad, dado que los nodos que se han encontrado más recientemente tienen la prioridad más alta. Si se utiliza id por prioridad se obtiene una búsqueda en anchura, porque los nodos más antiguos tienen la prioridad más alta. Estas asignaciones de prioridad hacen que las colas de prioridad operen como si fueran pilas y colas.

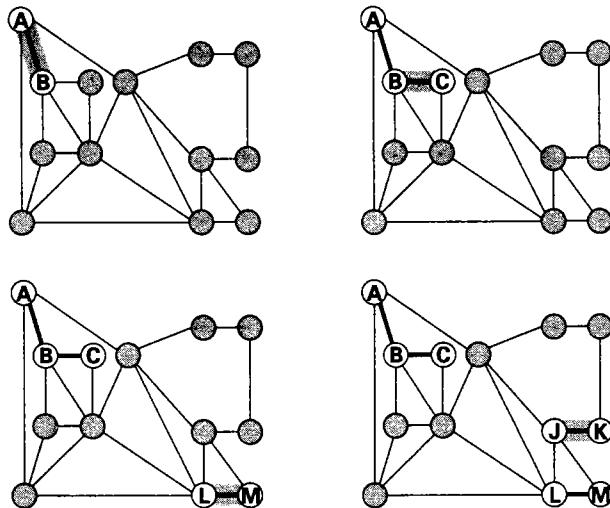


Figura 31.7 Pasos iniciales del algoritmo de Kruskal.

Método de Kruskal

Una aproximación del todo diferente para encontrar el árbol de expansión mínimo consiste simplemente en añadir aristas, una a una, utilizando en cada paso la arista más pequeña que no forme un ciclo. Dicho de otra manera, el algoritmo comienza con un bosque de árboles N : en N pasos combina dos árboles (utilizando la arista más pequeña posible) hasta que no exista mas que un árbol izquierdo. Este algoritmo data por lo menos de 1956 y se suele atribuir a J. Kruskal.

Las Figuras 31.7 y 31.8 muestran la operación de este algoritmo en el grafo del ejemplo. Las primeras aristas que se eligen son aquellas cuya longitud es la más pequeña (1) en el grafo. Después se ensayan las aristas de longitud 2; se observa, en particular, que FE se examina, pero no se incluye, dado que forma un ciclo con las aristas que ya se sabía que están en el árbol. Las componentes no conexas evolucionan progresivamente hacia un árbol, en contraste con la búsqueda en primera prioridad, en la que el árbol «engorda» arista a arista.

La implementación del algoritmo de Kruskal se puede ir componiendo a base de programas que ya se han estudiado. Primero se necesita considerar las aristas, una a una, por orden creciente de sus pesos. Una posibilidad sería la de ordenarlas simplemente, pero parece más conveniente utilizar una cola de prioridad, sobre todo porque no se necesita examinar todas las aristas. Esto se presenta con más detalle posteriormente. A continuación se necesita poder comprobar si una arista dada, al añadirse a las otras que ya se han cogido, en-

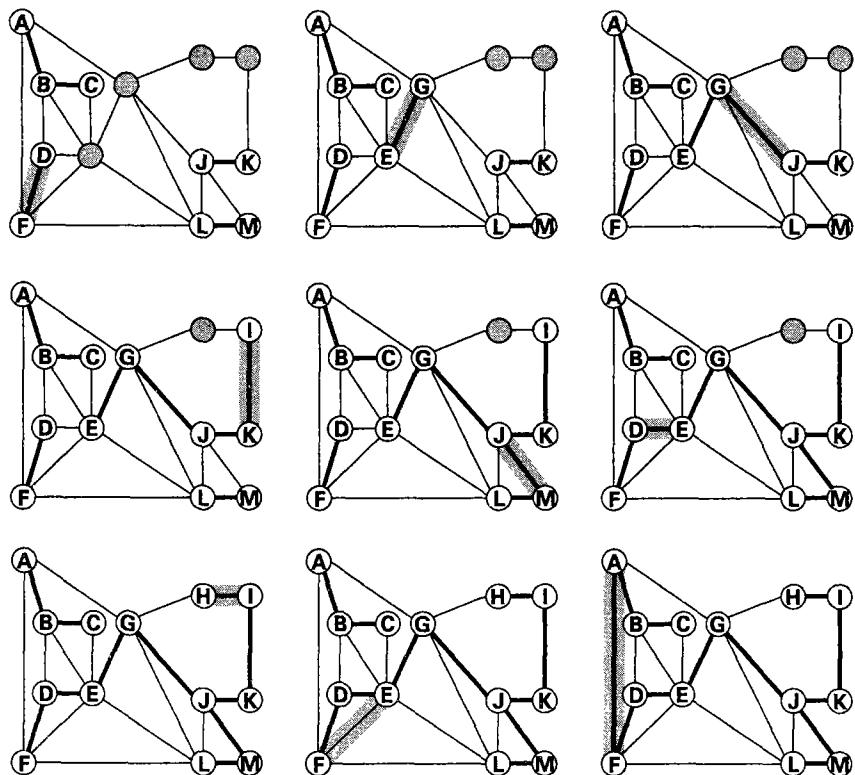


Figura 31.8 Final del algoritmo de Kruskal.

gendra un ciclo. Las estructuras de unión-pertenencia que se presentaron en el capítulo anterior están diseñadas precisamente para esta tarea.

Ahora, la estructura de datos apropiada para el grafo es tan sólo un array e con una entrada por cada arista. Éste se puede construir fácilmente a partir de la representación por listas de adyacencia o por matriz de adyacencia del grafo con una búsqueda en profundidad o algún procedimiento más simple. Sin embargo, en el programa siguiente se llena el array directamente a partir de los datos. Se utiliza la clase diccionario cola de prioridad CP del Capítulo 11, con las prioridades de los campos de ponderación del array e y una clase de equivalencia EQ basada en el procedimiento pertenencia del Capítulo 30, para verificar los ciclos. El programa llama simplemente al procedimiento aristaencontrada para cada arista del árbol de expansión; con un poco más de trabajo se podrían construir un array padre u otra representación.

```
void kruskal()
{
    int i, m, V, A;
```

```

struct arista
    { char v1, v2; int w; };
struct arista e[maxA];
CP cp(maxA); EQ eq(maxA);
cin >> V >> A;
for (i = 1; i >= A; i++)
    cin >> e[i].v1 >> e[i].v2 >> e[i].w;
for (i = 1; i >= A; i++)
    cp.insertar(i, e[i].w);
for (i = 0; i < V-1; )
{
    if (cp.vacio()) break; else m = cp.eliminar();
    if (cp.encontrar(indice(e[m].v1), indice(e[m].v2,1)))
        { cout << e[m].v1 << e[m].v2 << ' '; i++; };
}
}

```

El proceso puede terminar de dos formas. Si se encuentran $V-1$ aristas, entonces se tiene un árbol y se puede parar. Si en primer lugar se vacía la cola de prioridad, entonces hay que examinar todas las aristas sin encontrar el árbol de expansión; esto sucederá si el grafo es no conexo. El tiempo de ejecución de este programa está dominado por el consumo de tiempo al procesar las aristas de la cola de prioridad.

Propiedad 31.3 *El algoritmo de Kruskal construye el árbol de expansión mínimo de un grafo en $O(A \log A)$ pasos.*

La exactitud de este algoritmo se deriva también de la propiedad 31.1. Los dos conjuntos de vértices en cuestión son los conectados a las aristas elegidas para el árbol y los que todavía no se han tocado. Cada arista que se añade es la más pequeña de las que enlaza vértices de los dos conjuntos. El peor caso es un grafo que es no conexo, en el que se debe examinar todas las aristas. Incluso en un grafo conexo el peor caso seguirá siendo el mismo, porque el grafo puede estar compuesto de dos agrupamientos de vértices, todos ellos conectados por aristas muy pequeñas, y con una única arista muy grande que conecta a los dos agrupamientos. Entonces la arista más grande del grafo está en el árbol de expansión mínimo, pero será la última en salir de la cola de prioridad. Para grafos representativos, se debe esperar a que el árbol de expansión esté completo (tiene solamente $V-1$ vértices) antes de obtener la arista más grande del grafo, pero la construcción inicial de la cola de prioridad llevará siempre un tiempo proporcional a A (véase la propiedad 11.2).■

La Figura 31.9 muestra la construcción de un gran árbol de expansión mí-

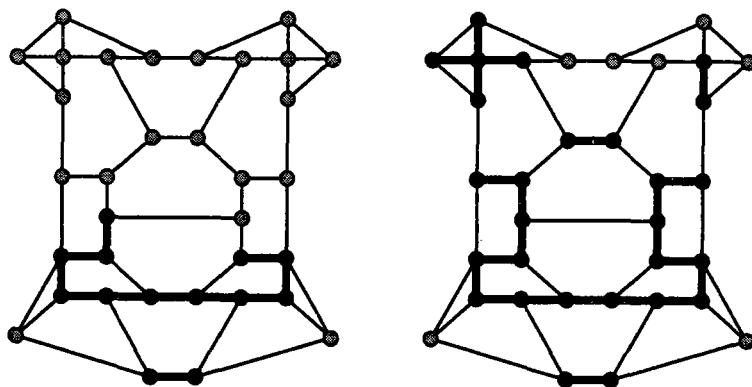


Figura 31.9 Construcción de un gran árbol de expansión mínimo con el algoritmo de Kruskal.

nimo con el algoritmo de Kruskal. Este diagrama muestra claramente cómo el método selecciona en primer lugar a todas las aristas pequeñas: el método añadirá las aristas más largas (diagonales) en último lugar.

En vez de utilizar colas de prioridad se podría simplemente ordenar las aristas por las ponderaciones iniciales y después tratarlas en orden. También, la verificación de ciclos puede hacerse en un tiempo proporcional a $A \log A$ con una estrategia mucho más simple que la de unión-pertenencia, lo que proporciona un algoritmo de árbol de expansión mínimo que siempre lleva $A \log A$ pasos. Éste es el método propuesto por Kruskal, pero en este libro se hace referencia a la versión modernizada anterior, que utiliza colas de prioridad y estructuras de unión-pertenencia, denominándola «algoritmo de Kruskal».

El camino más corto

El problema del *camino más corto* consiste en encontrar entre todos los caminos que conectan a dos vértices x e y dados de un grafo ponderado el que cumpla con la propiedad de que la suma de las ponderaciones de todas las aristas sea mínima.

Si las ponderaciones son todas igual a 1, entonces el problema sigue siendo interesante: ahora consiste en encontrar el camino que contenga al mínimo número de aristas que conecten a x e y . Además, ya se ha tratado un algoritmo que resuelve este problema: la búsqueda en anchura. Es fácil de demostrar por inducción que dicha búsqueda, si comienza en x , explorará en primer lugar todos los vértices que se puedan alcanzar desde x con una arista, después todos

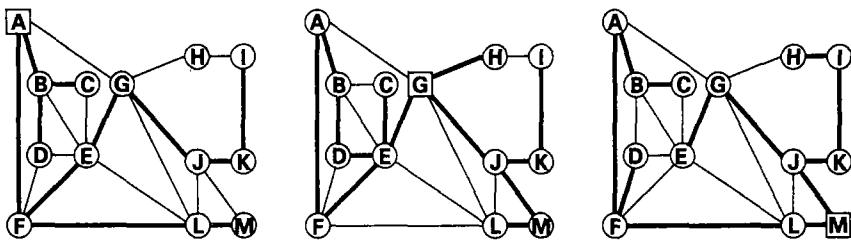


Figura 31.10 Árboles de expansión del camino más corto.

los vértices que se puedan alcanzar con dos aristas, etc., explorando todos los vértices que se puedan alcanzar con k aristas antes de encontrar alguno que necesite $k + 1$ aristas. Así, cuando se alcance y por primera vez, se ha encontrado el camino más corto desde x porque ningún otro camino más corto puede alcanzar y (véase la Figura 29.14).

En general, el camino desde x a y puede tocar a todos los vértices, por lo que normalmente se considera el problema de encontrar el más corto de los caminos que conectan a un vértice dado x con *todos* los otros vértices del grafo. Una vez más sucede que el problema es fácil de resolver con el algoritmo de recorrido de grafo en primera prioridad dado anteriormente.

Si se dibuja el camino más corto desde x a todos los otros vértices del grafo, entonces se obtiene claramente que no hay ciclos, y se tiene un árbol de expansión. Cada vértice conduce a un árbol de expansión diferente; por ejemplo, la Figura 31.10 muestra los árboles de expansión de los caminos más cortos para los vértices A, G y M del grafo de ejemplo que se ha estado utilizando.

La solución de la búsqueda en primera prioridad para este problema es virtualmente idéntica a la solución del árbol de expansión mínimo: se construye el árbol por el vértice x añadiendo a cada paso el vértice del margen que sea el más próximo a x (antes, se añade el más próximo al *árbol*). Para encontrar cuál de los vértices del margen es el más próximo a x , se utiliza el array val : para cada vértice k del árbol, $\text{val}[k]$ es la distancia entre ese vértice y x , utilizando el camino más corto (que debe estar formado por los nodos del árbol). Cuando se añade k al árbol, se actualiza el margen recorriendo la lista de adyacencia de k . Para cada nodo t de la lista, la distancia más corta de x a $t \rightarrow v$ a través de k es $\text{val}[k] + t \rightarrow v$. Así, el algoritmo se implementa de forma trivial, utilizando esta cantidad en prioridad en el programa de recorrido de grafos en primera prioridad.

La Figura 31.11 muestra los primeros cuatro pasos de la construcción del árbol de expansión del camino más corto para el vértice A del ejemplo. Primero se explora el vértice más próximo a A, que es B. C y F están a la distancia 2 de A, por lo que se exploran a continuación (en el orden que la cola de prioridad devuelve para ellos, en este caso C y después F). El final del proceso se muestra en la Figura 31.12 y el contenido de la cola de prioridad durante las búsquedas se muestra en la Figura 31.13.

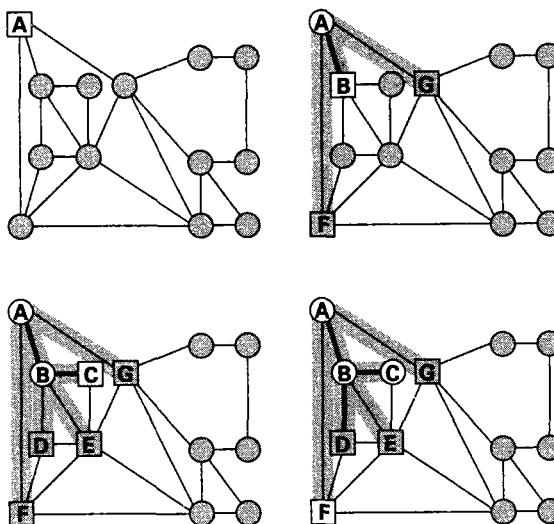


Figura 31.11 Pasos iniciales de la construcción de un árbol de camino más corto.

A continuación, D puede conectarse a F o a B para obtener un camino de distancia 3 a A. (El algoritmo conecta D a B porque B se puso en el árbol antes que F; así, D estaba ya en el margen cuando F se puso en el árbol, y F no proporciona un camino más corto hacia A.) Despues se añaden al árbol L E M G J K I, en orden creciente, de su distancia mínima a A. Así, por ejemplo, H es el nodo más lejano desde A: el camino AFEGH tiene un peso total de 8. No existe el camino más corto hacia H, y el camino más corto desde A a todos los otros nodos no es el más largo.

La Figura 31.14 muestra los valores finales de los arrays `padre` y `val` para el ejemplo. Así el camino más corto desde A a H tiene un peso total de 8 (encontrado en `val[8]`, la entrada para H) y va desde A a F a E a G y a H (encontrado leyendo al revés en el array `padre`, comenzando en H). Obsérvese que este programa depende de que la entrada de `val` para la raíz sea cero, la convención que se adoptó para `listasbpp`.

Propiedad 31.4 *La búsqueda en primera prioridad en un grafo disperso calcula el árbol de camino más corto en $O((A + V)\log V)$ pasos.*

La demostración de este algoritmo se puede hacer de forma similar a la propiedad 31.1. Con una cola de prioridad implementada, utilizando montículos como en el Capítulo 12, la búsqueda en primera prioridad puede *siempre* asegurarse que funcionará en el número máximo de pasos mencionado, sin importar qué regla de prioridad se utilice. ■

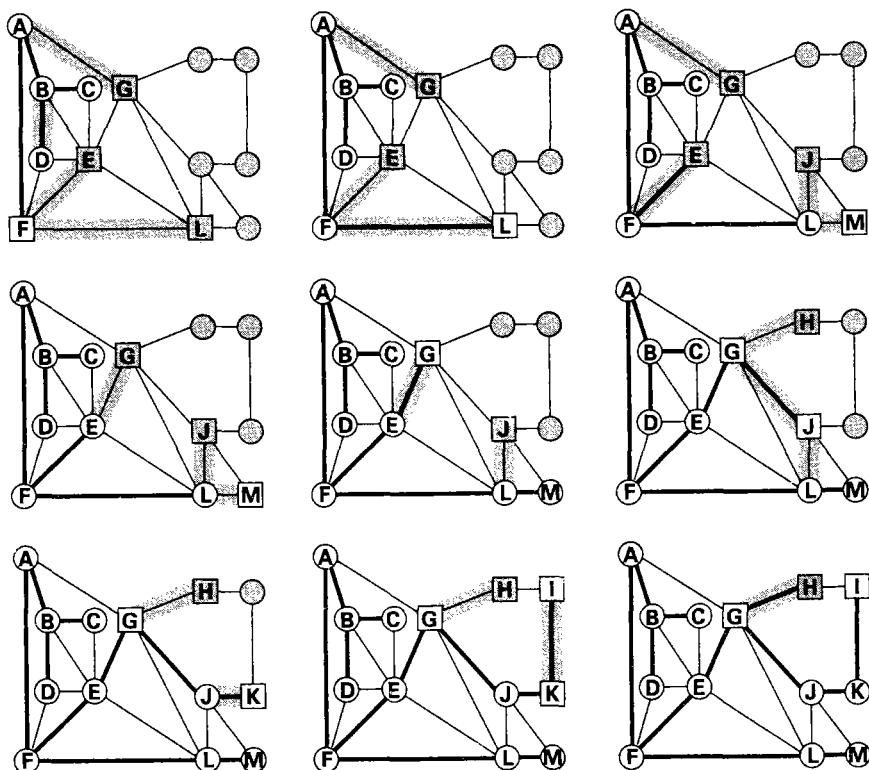


Figura 31.12 Final de la construcción de un árbol de camino más corto.

Más adelante se verá cómo una implementación diferente de la cola de prioridad puede dar un algoritmo en V^2 que es apropiado para los grafos densos. Para el problema del camino más corto, esto se reduce a un método que data al menos de 1959 y que suele atribuirse a E. Dijkstra.

La Figura 31.15 muestra un árbol de camino más corto de gran tamaño. Como antes, las longitudes de las aristas se utilizan en este grafo como pesos, por lo que la solución llega a encontrar el camino de longitud mínima desde el

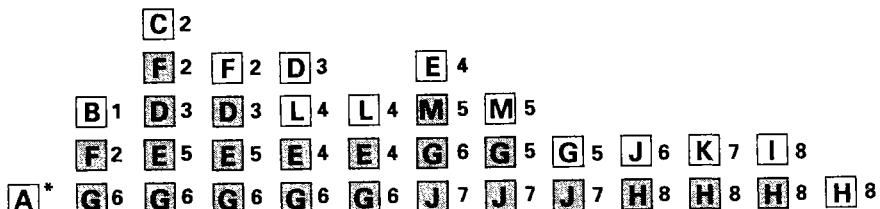


Figura 31.13 Contenido de la cola de prioridad durante la construcción del árbol de camino más corto.

k	1	2	3	4	5	6	7	8	9	10	11	12	13
nombre(padre[k])	A	B	B	F	A	E	G	K	G	I	F	L	
val[k]	0	1	2	3	4	2	5	8	8	6	7	4	5

Figura 31.14 Representación del árbol de expansión del camino más corto.

nodo inferior izquierdo a todos los demás. Posteriormente, se presentará una mejora que puede ser apropiada para tales grafos. Pero incluso en este grafo podría ser apropiado utilizar otros valores para los pesos: por ejemplo, si este grafo representa un laberinto (véase el Capítulo 29), el peso de una arista puede representarse por la distancia del propio laberinto, no por los «atajos» dibujados en el grafo.

Árbol de expansión mínimo y camino más corto en grafos densos

Para un grafo representado por una matriz de adyacencia, lo mejor es utilizar una representación por array no ordenado para la cola de prioridad, de manera que se obtenga un tiempo de ejecución V^2 para cualquier algoritmo de recorrido del grafo en primera prioridad. Esto se hace combinando el bucle de actualización de las prioridades y el de encontrar el mínimo: cada vez que se mueve un vértice del margen, se pasa a través de todos los vértices, actualizando sus prioridades si es necesario y siguiendo la pista del valor mínimo encontrado. Esto proporciona un algoritmo lineal para la búsqueda en primera prioridad en gra-

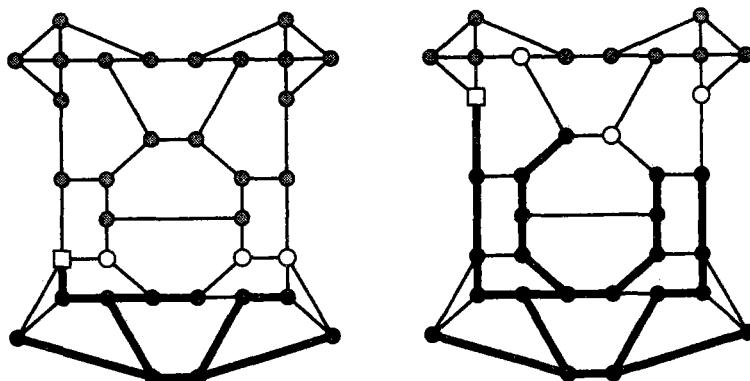


Figura 31.15 Construcción de un gran árbol del camino más corto.

fos densos (y también para los problemas del árbol de expansión mínimo y el camino más corto).

Especificamente, se gestiona la cola de prioridad en el array `val` (esto también se puede hacer en `listasbpp`, como se presentó antes), pero se implementan las operaciones de cola de prioridad directamente en vez de utilizar montículos. Como antes, el signo de las entradas de `val` indica si el vértice correspondiente está en el árbol o en la cola de prioridad. Todos los vértices comienzan en la cola de prioridad y tienen la prioridad del centinela `novisto`. Para cambiar la prioridad de un vértice, simplemente se asigna la nueva prioridad en la entrada de `val` de este vértice. Para eliminar el vértice de prioridad más alta, se explora a través del array `val` para encontrar el vértice con el mayor valor negativo (el más próximo a 0), y entonces se toma su complemento en `val`. Después de hacer estas modificaciones mecánicas en el programa `listasbpp` que se ha estado utilizando, se obtiene el programa compacto siguiente:

```
void buscar() // BPP, matriz de adyacencia
{
    int k, t, min = 0;
    for (k = 1; kn<= V; k++)
        { val[k] = novisto; padre[k] = 0; }
    val[0] = novisto-1;
    for (k = 1; k != 0; k = min, min = 0)
    {
        val[k] = -val[k];
        if (val[k] == -novisto) val[k] = 0;
        for (t = 1; t <= V; t++)
            if (val[t] < 0)
            {
                if (a[k][t] && (val[t] < -prioridad))
                    { val[t] = -prioridad; padre[t] = k; }
                if (val[t] > val[min]) min = t;
            }
    }
}
```

Es preciso señalar que se ha utilizado un valor superior en una unidad a `-novisto`, como un centinela para encontrar el mínimo, y que el opuesto de dicho valor debe ser representable.

Si se almacenan los pesos en la matriz de adyacencia y se utiliza `a[k][t]` para prioridad en este programa, se obtiene el algoritmo de Prim para la construcción del árbol de expansión mínimo; si se utiliza `val[k]+a[k][t]` para prioridad se obtiene el algoritmo de Dijkstra para el problema del camino más corto. Como antes, si se incluye el código para que `id` indique el número de

vértices explorados y se utiliza $V - id$ para prioridad, se obtiene la búsqueda en profundidad. Este programa sólo se diferencia del de la búsqueda en primera prioridad, con el que se ha estado trabajando para los grafos dispersos, en la representación del grafo que se utiliza (matriz de adyacencia en vez de listas de adyacencia) y en la implementación de la cola de prioridad (array no ordenado en vez de montículo indirecto).

Propiedad 31.5 *Los problemas del árbol de expansión mínimo y del camino más corto pueden resolverse en tiempo lineal para grafos densos.*

De la inspección del programa se deduce inmediatamente que el tiempo de ejecución del peor caso es proporcional a V^2 . Cada vez que se explora un vértice, un recorrido de las V entradas de una fila de la matriz de adyacencia permite cumplir el doble objetivo de verificar todas las aristas adyacentes, actualizar la cola de prioridad y encontrar el valor mínimo que contiene. Así, el tiempo de ejecución es lineal cuando A es proporcional a V^2 .■

Se han presentado tres programas para el problema del árbol de expansión mínimo con diferentes características de rendimiento: el método de la búsqueda en primera prioridad, el algoritmo de Kruskal y el algoritmo de Prim. El algoritmo de Prim es posiblemente el más rápido de los tres para algunos grafos, el de Kruskal para otros y el método de búsqueda en primera prioridad para otros. Como se dijo anteriormente, el peor caso para el método de búsqueda en primera prioridad es $(A + V)\log V$, mientras que el peor caso para el de Prim es V^2 y para el de Kruskal es $A \log A$. Pero sería poco prudente elegir entre los algoritmos sobre la base de estas fórmulas porque es improbable que «el peor caso» suceda en la práctica. De hecho, el método de la búsqueda en prioridad y el de Kruskal son ambos susceptibles de ejecutarse en tiempo proporcional a A para los grafos que aparecen normalmente en la práctica: el primero, porque la mayoría de las aristas no necesitan una actualización de la cola de prioridad, que lleva $\log V$ pasos, y el segundo porque es probable que la arista de mayor longitud del árbol de expansión mínimo sea lo suficientemente pequeña como para que no se extraigan muchas aristas de la cola de prioridad. Por supuesto, el método de Prim también se ejecuta en un tiempo proporcional a aproximadamente A en grafos densos (pero no debe utilizarse en grafos dispersos).

Problemas geométricos

Supóngase que se tienen N puntos dados de un plano y que se desea encontrar el conjunto de segmentos de longitud más pequeña de los que conectan a todos los puntos. Éste es un problema geométrico denominado el problema del *árbol de expansión mínimo euclíadiano*, que se puede resolver utilizando el algoritmo

para grafos dado anteriormente, pero parece claro que la geometría del mismo proporciona suficiente estructura extra como para que se puedan desarrollar algoritmos mucho más eficaces.

El medio para resolver el problema euclíadiano utilizando el algoritmo anterior consiste en construir un grafo completo con N vértices y $N(N - 1)/2$ aristas, cada una de las cuales conecta cada par de vértices ponderados por la distancia entre los puntos correspondientes. Entonces se puede construir el árbol de expansión mínimo por medio del algoritmo de Prim en un tiempo proporcional a N^2 .

Se ha demostrado que es posible hacerlo mejor. En efecto, la estructura geométrica del problema hace irrelevantes a la mayor parte de las aristas, y se pueden eliminar una mayoría de ellas antes de comenzar a construir el árbol de expansión mínimo. De hecho, también se ha demostrado que el árbol de expansión mínimo es un subconjunto del grafo que se obtiene al elegir solamente las aristas a partir del dual del diagrama de Voronoi (véase el Capítulo 28). Se sabe que este grafo tiene un número de aristas proporcional a N y que tanto el algoritmo de Kruskal como el método de búsqueda en primera prioridad funcionan eficazmente en tales grafos dispersos. En principio, podría calcularse el dual del diagrama de Voronoi (lo que lleva un tiempo proporcional a $N \log N$), y a continuación ejecutar o bien el algoritmo de Kruskal o bien el método de búsqueda en primera prioridad para obtener el algoritmo del árbol de expansión mínimo euclíadiano que se ejecuta en un tiempo proporcional a $N \log N$. Pero la escritura de un programa para calcular el dual de Voronoi es más bien un desafío, incluso para programadores experimentados, y por ello esta aproximación al problema es probablemente impracticable.

Otro enfoque, que se puede utilizar en conjuntos de puntos aleatorios, consiste en aprovecharse de la distribución de puntos para limitar el número de aristas incluidas en el grafo, como en el método de la rejilla que se utilizó en el Capítulo 26 para la búsqueda por rango. Si se divide el plano en cuadrados de tal forma que cada uno de ellos contenga aproximadamente $\lg N/2$ puntos, y entonces se incluyen en el grafo sólo aquellas aristas que conecten cada punto de los situados en cuadrados vecinos, entonces es muy probable (pero no garantizado) obtener todas las aristas del árbol de expansión mínimo, lo que significaría que el algoritmo de Kruskal o el método del árbol de búsqueda en primera prioridad acabarían el trabajo eficazmente.

Es interesante hacer una reflexión sobre las relaciones entre los grafos y los algoritmos geométricos, que se hará dado a conocer por el problema expuesto en los párrafos anteriores. Es cierto que muchos problemas pueden formularse bien como problemas geométricos, bien como problemas de grafos. Si el emplazamiento físico real de los objetos es una característica dominante, entonces pueden ser apropiados los algoritmos geométricos de los Capítulos 24-28; pero si las interconexiones entre objetos tienen una importancia fundamental, entonces podrán ser mejores los algoritmos sobre grafos de esta sección. El árbol de expansión mínimo euclíadiano parece ser la interfaz entre estos dos enfoques (los datos de entrada afectan la geometría y los de salida, las interconexiones),

y el desarrollo de métodos simples y directos para éste y otros problemas relacionados con él continúa siendo una meta elusiva.

Otro lugar donde los algoritmos grafos y geométricos interactúan es en el problema de encontrar el camino más corto entre x e y en un grafo cuyos vértices sean puntos del plano y cuyas aristas sean líneas que los conectan. El grafo laberinto que se ha utilizado puede considerarse como un grafo de este tipo. La solución a este problema es simple: utilizar la búsqueda en primera prioridad, dando como prioridad de cada vértice del margen que se encuentre la distancia, en el árbol, entre x y el vértice del margen, más la distancia euclíadiana entre dicho vértice del margen e y . A continuación se para cuando y se ha añadido al árbol. Este método encontrará rápidamente el camino más corto desde x a y yendo siempre en la dirección de y , mientras que el algoritmo sobre el grafo estándar tiene que «buscar» y . Desplazarse de un extremo a otro de un gran grafo laberinto puede necesitar que se examine un número de nodos proporcional a \sqrt{V} , mientras que el algoritmo estándar debe examinar prácticamente todos los nodos.

Ejercicios

1. Encontrar otro árbol de expansión mínimo para el grafo ejemplo del principio del capítulo.
2. Presentar un algoritmo para encontrar el *bosque de expansión mínimo* de un grafo conexo (cada vértice debe ser alcanzado por alguna arista, pero no es necesario que el grafo resultante sea conexo).
3. ¿Existe un grafo con V vértices y A aristas para el que la solución en primera prioridad del problema del árbol de expansión mínimo pueda necesitar un tiempo proporcional a $(A + V)\log V$? Dar un ejemplo o explicar la respuesta.
4. Supóngase que se mantiene la cola de prioridad por medio de una lista ordenada en las implementaciones de recorrido de grafos en general. ¿Cuál podrá ser el tiempo de ejecución del peor caso para un factor constante? ¿Cuándo podría ser apropiado el método, si lo es alguna vez?
5. Presentar contraejemplos que muestren por qué la siguiente estrategia «insaciable» no resuelve ni el problema del camino más corto ni el del árbol de expansión mínimo: «en cada paso se exploran los vértices inexplorados que sean los más próximos al que se acaba de explorar».
6. Presentar los árboles del camino más corto para los otros nodos del grafo del ejemplo.
7. Describir cómo se podría encontrar el árbol de expansión mínimo de un grafo extremadamente grande (demasiado grande como para poderse almacenar en la memoria principal).
8. Escribir un programa para generar grafos aleatorios conexos con V vértices, y después encontrar el árbol de expansión mínimo y el del camino más corto

para algunos vértices. Utilizar pesos aleatorios entre 1 y V . ¿Qué relación existe entre los pesos de los árboles y los diferentes valores de V ?

9. Escribir un programa para generar grafos aleatorios completos y ponderados con V vértices tan sólo llenando una matriz de adyacencia con números aleatorios entre 1 y V . Determinar empíricamente qué método encuentra el árbol de expansión mínimo de forma más rápida para $V = 10, 25, 100$: el de Prim o el de Kruskal.
10. Presentar un contraejemplo para mostrar por qué el método siguiente no sirve para encontrar el árbol de expansión mínimo euclíadiano: «ordenar los puntos por sus coordenadas x , a continuación encontrar los árboles de expansión mínimos de la primera mitad, de la segunda mitad, y finalmente encontrar la arista más corta que los conecta».

Grafos dirigidos

Los grafos dirigidos son aquellos en los que las aristas que conectan los nodos son de sentido único; esta estructura adicional hace más difícil la determinación de ciertas propiedades. El procesamiento de este tipo de grafos es comparable a conducir por una ciudad con muchas calles de sentido único o a viajar por un país cuyas líneas aéreas raramente tienen rutas de ida y vuelta: ir de un punto a otro en estas condiciones puede ser un desafío.

A menudo, la dirección de las aristas refleja algún tipo de relación de precedencia en la aplicación que se está modelando. Por ejemplo, un grafo dirigido puede utilizarse como modelo para una cadena de fabricación: los nodos corresponden a las tareas a realizar, y si existe una arista entre el nodo x y el nodo y significa que la tarea correspondiente al nodo x debe hacerse antes que la correspondiente al nodo y . ¿Cómo se debe decidir cuándo llevar a cabo cada tarea de tal forma que no se viole ninguna de estas relaciones de precedencia?

En este capítulo se estudiará la búsqueda en profundidad en los grafos dirigidos, así como los algoritmos para procesar la *clausura transitiva* (que resume la información de conectividad), el método de *ordenación topológica* y el de las *componentes fuertemente conexas* (que están ligadas con las relaciones de precedencia).

Como se mencionó en el Capítulo 29, las representaciones de los grafos dirigidos son simples extensiones (realmente, restricciones) de las de los grafos no dirigidos. En la representación por lista de adyacencia, cada arista aparece solamente una vez: la arista de x hacia y se representa como un nodo de la lista que contiene a y en la lista enlazada correspondiente a x . En la representación por matriz de adyacencia, se necesita mantener una matriz V^*V completa, con un bit 1 en la fila x -columna y (pero no necesariamente en la fila y -columna x) si existe una arista de x hacia y .

En la Figura 32.1 se muestra un grafo dirigido similar al no dirigido que se ha estado considerando. Este grafo está constituido por las aristas AG AB CA LM JM JL JK ED DF HI FE AF GE GC HG GJ LG IH ML. Ahora, el orden en el que aparecen los vértices al especificar las aristas tiene mucha importan-

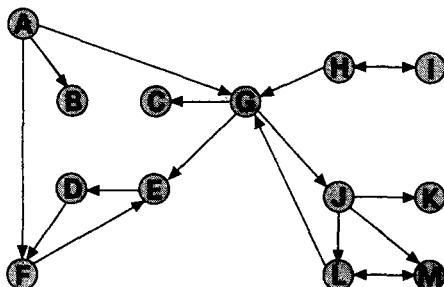


Figura 32.1 Un grafo dirigido.

cia: la notación AG describe una arista que apunta de A hacia G, pero *no* de G hacia A. Aun así, es posible tener dos aristas entre dos nodos, una en cada dirección (en la Figura 32.1 existen HI e IH así como LM y ML).

No es posible percibir en estas representaciones la diferencia entre un grafo dirigido y uno no dirigido cuando cada arista tiene las dos direcciones opuestas. Así, algunos de los algoritmos de este capítulo pueden considerarse como una generalización de los algoritmos de los capítulos anteriores.

Búsqueda en profundidad

El algoritmo de búsqueda en profundidad del Capítulo 29 puede utilizarse exactamente igual con los grafos dirigidos. De hecho, opera de una manera un poco más directa que con los grafos no dirigidos dado que no necesita tener en cuenta las dobles aristas entre nodos, a menos que estén incluidas explícitamente en el grafo. Sin embargo, los árboles de búsqueda tienen una estructura algo más compleja. Por ejemplo, la Figura 32.2 muestra la estructura de búsqueda en profundidad que describe la acción del algoritmo recursivo del Capítulo 29 en el grafo del ejemplo. Igual que antes, ésta es otra forma de dibujar el grafo: las aristas en trazo continuo corresponden a aquellas que realmente se han utilizado para explorar los vértices por medio de llamadas recursivas, y las aristas de línea de puntos corresponden a aquellas que apuntan a vértices que ya han sido explorados en el momento en el que se consideró a la arista. Los nodos se exploran en el orden A F E D B G J K L M C H I.

Las direcciones de las aristas hacen que el bosque de búsqueda en profundidad sea muy diferente del correspondiente a un grafo no dirigido. Por ejemplo, a pesar de que el grafo original sea conexo, no lo es la estructura de bús-

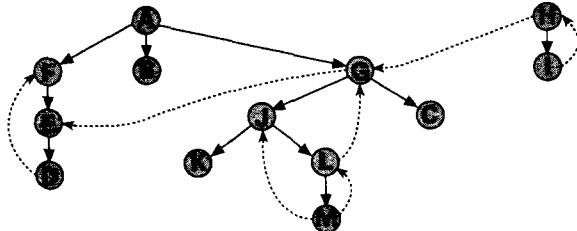


Figura 32.2 Bosque de búsqueda en profundidad para un grafo dirigido.

queda en profundidad definida por las aristas de trazo continuo: es un bosque, no un árbol.

Para los grafos no dirigidos se tiene solamente un tipo de aristas de línea de puntos, las que conectan a un vértice con algún ascendiente en el árbol. Para los grafos dirigidos existen tres tipos de aristas de línea de puntos: las aristas *hacia arriba*, que apuntan desde un vértice hacia alguno de sus ascendientes en el árbol, las aristas *hacia abajo*, que apuntan desde un vértice hacia alguno de sus descendientes en el árbol, y las aristas *transversales*, que apuntan desde un vértice hacia otro cualquiera que no es ni ascendiente ni descendiente del primero en el árbol.

Como en el caso de los grafos no dirigidos, existe gran interés en conocer las propiedades de conectividad de los grafos dirigidos. Debería ser posible responder a preguntas tales como «¿Existe un *camino dirigido* desde el vértice x al vértice y (un camino que siga las aristas solamente en la dirección indicada)?» y «¿Qué vértices son accesibles desde el vértice x por medio de un camino dirigido?» y «¿Existe un camino dirigido desde el vértice x al vértice y y un camino dirigido desde y a x ?». De igual forma que con los grafos no dirigidos, será posible contestar a tales preguntas modificando adecuadamente el algoritmo básico de búsqueda en profundidad; sin embargo, los diferentes tipos de aristas de línea de puntos hacen que estas modificaciones sean algo más complejas.

Clausura transitiva

En los grafos no dirigidos, la conectividad simple proporciona los vértices que pueden ser alcanzados desde un vértice dado recorriendo las aristas del grafo: todos ellos son de la misma componente conexa. De forma similar, para los grafos dirigidos, a menudo se tiene interés en conocer el conjunto de vértices que

pueden ser alcanzados desde un vértice dado recorriendo las aristas del grafo en la dirección indicada. El problema es mucho más complejo en los grafos dirigidos que en la conectividad simple.

Es fácil demostrar que el procedimiento recursivo `visitar` del método de búsqueda en profundidad del Capítulo 29 explora todos los nodos que se pueden alcanzar desde el nodo de partida. Así pues, si se modifica el procedimiento, de forma que imprima los nodos que explora (es decir, insertando `cout << nombre(k)`), se obtendrá la lista de todos los nodos que se pueden alcanzar desde el nodo de partida. Pero es preciso tener en cuenta que *no* es necesariamente verdad que todo árbol del bosque de búsqueda en profundidad contiene a todos los nodos que se puedan alcanzar desde su propia raíz: en el ejemplo, todos los nodos del grafo se pueden alcanzar desde H, y no solamente I. Para obtener todos los nodos que se pueden explorar desde cada uno de ellos, es suficiente con llamar V veces a `visitar`, una para cada nodo:

```
for (k = 1; k <= V; k++)
{
    id = 0;
    for (j = 1; j <= V; j++) val[j] = 0;
    visitar(k);
    cout >> '\n';
}
```

Este programa genera la siguiente salida para el grafo dirigido de la Figura 32.1. Como anteriormente, el orden de los nombres de los vértices de cada línea es un efecto de la representación particular del grafo y del procedimiento de búsqueda elegido. El *conjunto* de nodos de cada línea es una propiedad estructural del propio grafo: en cada línea todos los nodos se pueden alcanzar por medio de un camino dirigido desde el primer nodo de la línea.

A	F	E	D	B	G	J	K	L	M	C
B										
C	A	F	E	D	B	G	J	K	L	M
D	F	E								
E	D	F								
F	E	D								
G	J	K	L	M	C	A	F	E	D	B
H	G	J	K	L	M	C	A	F	E	D
I	H	G	J	K	L	M	C	A	F	E
J	K	L	G	C	A	F	E	D	B	M
K										
L	G	J	K	M	C	A	F	E	D	B
M	L	G	J	K	C	A	F	E	D	B

Para los grafos no dirigidos este programa generaría una tabla con la caracterís-

tica de que toda línea que corresponde a un nodo de una componente conexa contiene todos los nodos de dicha componente. La tabla anterior tiene una propiedad similar: ciertas líneas contienen conjuntos de nodos idénticos. Más adelante se examinará la generalización de la noción de conectividad que explica esta propiedad.

Como es habitual, se podrá incluir alguna instrucción para hacer un procedimiento extra que proporcione algo más que la impresión de dicha tabla. Una operación que podría realizarse es añadir una arista dirigida de x hacia y si existe algún camino para ir de x a y . El grafo que resulta de añadir todas las aristas de este tipo a un grafo dirigido se denomina la *clausura transitiva* del mismo. Por lo regular, se añaden un gran número de aristas y la clausura transitiva probablemente será densa, por lo que se prefiere una representación por matriz de adyacencia. Esto es una analogía de las componentes conexas de un grafo no dirigido: una vez que se ha llevado a cabo esta operación, se puede responder rápidamente a preguntas tales como «¿Existe un medio de ir desde x hacia y ?».

Propiedad 32.1 *La búsqueda en profundidad se puede utilizar para construir la clausura transitiva de un grafo dirigido en $O(V(A+V))$ pasos para un grafo disperso y en $O(V^3)$ pasos para un grafo denso.*

Esto proviene directamente de las propiedades básicas del Capítulo 29: se lleva a cabo una búsqueda en profundidad para cada uno de los V vértices del grafo. El mismo resultado es válido para la búsqueda en amplitud: como se mencionó con anterioridad, el orden en el que se exploran los nodos no es particularmente relevante en este problema.■

Existe un programa no recursivo notablemente simple para construir la clausura transitiva de un grafo representado por una matriz de adyacencia:

```
for ( y = 1; y <= V; y++)
    for ( x = 1; x <= V; x++)
        if (a[x][y])
            for ( j = 1; j <= V; j++)
                if (a[y][j]) a[x][j] = 1;
```

S. Warshall inventó este método en 1962, utilizando la simple observación de que «si existe un medio de ir del nodo x al nodo y y un medio de ir del nodo y al nodo j , entonces existe un medio de ir del nodo x al nodo j ». La idea es hacer algo más sólida esta observación, de forma que se pueda hacer el cálculo en una sola pasada a través de la matriz, diciendo: «si existe un medio de ir del nodo x al nodo y utilizando solamente los nodos con índices menores que y y un medio de ir del nodo y al nodo j , entonces existe un medio de ir del nodo x al nodo j utilizando solamente los nodos con índices menores que $y+1$ ». El programa anterior es una implementación directa de esto.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	0	0	0	1	1	0	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	1	1	1	0	0	0	0	0
I	0	0	0	0	0	0	1	1	0	0	0	0	0
J	0	0	0	0	0	0	0	1	1	1	1	1	1
K	0	0	0	0	0	0	0	0	1	0	0	0	0
L	0	0	0	0	0	1	0	0	0	1	1	1	1
M	0	0	0	0	0	0	0	0	0	0	1	1	1

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	0	1	1	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	1	1	1	1	0	0	1	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1	0	0	1	1	1	1
H	0	0	0	1	1	1	1	1	1	1	1	1	1
I	0	0	0	1	1	1	1	1	1	1	1	1	1
J	0	0	0	0	0	0	0	1	1	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	0	0	0
L	0	0	0	0	1	1	1	1	1	1	1	1	1
M	0	0	0	0	0	0	0	0	0	0	1	1	1

Figura 32.3 Primeras etapas del algoritmo de Warshall.

El método de Warshall convierte la matriz de adyacencia de un grafo en la matriz de adyacencia de su clausura transitiva. Un medio de seguir al algoritmo es imaginar cómo pone los valores en la matriz de adyacencia, línea a línea. El tratamiento de la columna y consiste en reemplazar cada línea que tiene un bit 1 en la columna y por su propio «o» y de la línea y. La Figura 32.3 muestra la matriz inicial del grafo ejemplo y el estado de la misma, una vez que se han tratado las dos primeras columnas y la mitad de la tercera: en este punto sólo está afectada la línea C. La Figura 32.4 muestra la matriz antes de que se hayan tratado las últimas columnas y el resultado final (la clausura transitiva).

Propiedad 32.2 *El algoritmo de Warshall encuentra la clausura transitiva en $O(V^3)$ pasos.*

Si al principio la matriz está llena de bits 1, la propiedad se deduce simplemente de la inspección dados los tres lazos anidados. Pero incluso un grafo disperso

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	1	1	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	0	1	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	1	1	1	1	1	1	1	0	1	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	0	0	0	0	0	0	0	1	1	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	0	0	0
L	1	1	1	1	1	1	1	0	1	1	1	1	1
M	0	0	0	0	0	0	0	0	0	1	1	1	1

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	1	1	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	1	0	1	1	1	1
D	0	0	0	1	1	1	1	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	0	0	0	1	1	1	1	0	0	0	0	0	0
G	1	1	1	1	1	1	1	1	0	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	0	0	0	0	0	0	0	0	0	0	1	1	1
K	0	0	0	0	0	0	0	0	0	0	0	1	0
L	1	1	1	1	1	1	1	1	1	1	1	1	1
M	0	0	0	0	0	0	0	0	0	0	1	1	1

Figura 32.4 Etapas finales del algoritmo de Warshall.

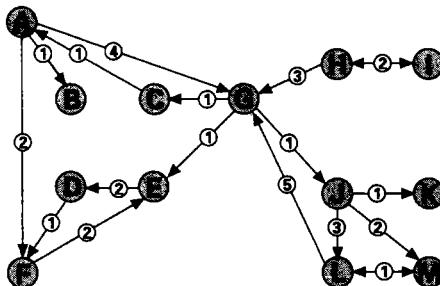


Figura 32.5 Un grafo dirigido ponderado.

puede reducirse rápidamente a este caso. Por ejemplo, si el primer nodo está conectado a todos los otros, entonces la matriz se encuentra llena de bits 1 antes de que y tome el valor 2. ■

Para grafos muy grandes, este cálculo puede organizarse de manera que las operaciones sobre los bits puedan hacerse palabra a palabra (de máquina), lo que conduce a considerables ganancias de tiempo en muchos entornos.

Todos los caminos más cortos

La clausura transitiva de un grafo no ponderado (dirigido o no) responde a la pregunta «¿Existe un camino desde x a y ?» para todos los pares de vértices x, y . Para los grafos ponderados (dirigidos o no) se puede desear construir una tabla que permita encontrar el camino más corto desde x a y para todos los pares de vértices. Éste es el *problema de todos los pares de caminos más cortos*. Por ejemplo, en el grafo ponderado, que se muestra en la Figura 32.5, se desea conocer con un simple acceso a esta tabla que el camino más corto de M a K tiene una longitud 8, y el camino más corto de J a F la tiene de 12, etcétera.

Como ya se ha dicho, el algoritmo del camino más corto del capítulo anterior encuentra el camino más corto desde el vértice de partida a cada uno de los otros vértices, por lo que se necesita solamente ejecutar este procedimiento V veces, comenzando una vez en cada vértice. Esto proporciona un algoritmo que se ejecuta en $O((A + V)V \log V)$ pasos. Pero también es posible utilizar un método parecido al de Warshall, que suele atribuirse a R. W. Floyd:

```
for (y = 1; y <= V; y++)
    for (x = 1; x <= V; x++)
```

```

if (a[x][y])
for (j = 1; j <= V; j++)
if (a[y][j] > 0)
if (!a[x][j] || (a[x][y]+a[y][j] < a[x][j]))
a[x][j] = a[x][y] + a[y][j];

```

La estructura del algoritmo es precisamente igual a la del método de Warshall. En lugar de utilizar «o» para seguir la pista de los caminos, se hace un pequeño cálculo para cada arista determinando si forma parte de un nuevo camino corto: «el camino más corto desde el nodo x hasta el nodo j utilizando solamente los nodos con índices menores que $y+1$ es, o bien el camino más corto desde el nodo x al j utilizando solamente los nodos con índices menores que y , o bien, si es más pequeño, el camino más corto desde x a y más la distancia de y a j ». Como es habitual, una entrada en la matriz de valor 0 corresponde a la ausencia de la arista indicada; el programa podría simplificarse un poco (suprimiendo todas las comparaciones con cero) utilizando un valor centinela correspondiente a una arista de peso infinitamente grande.

Propiedad 32.2 *El algoritmo de Floyd resuelve el problema de todos los pares de caminos más cortos en $O(V^3)$ pasos.*

Esta propiedad se deduce por los mismos razonamientos que la propiedad 32.1.■

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	1	0	0	0	0
H	0	0	0	0	0	3	0	1	0	0	0	0	0
I	0	0	0	0	0	0	1	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	1	3	2	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	5	0	0	0	0	1	0	0
M	0	0	0	0	0	0	0	0	0	1	0	0	0

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	0	0	0	3	5	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	0	1	0	0	0
H	0	0	0	0	0	3	0	1	0	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	3	2	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	5	0	0	0	0	0	1	0
M	0	0	0	0	0	0	0	0	0	0	0	0	1

Figura 32.6 Etapas iniciales del algoritmo de Floyd.

Las Figuras 32.6 y 32.7 muestran con mayor detalle el rendimiento del algoritmo de Floyd en el ejemplo, en los mismos instantes exactamente que en las Figuras 32.3 y 32.4, para poder comparar. Las entradas cero en las diferentes matrices, que corresponden a la ausencia de un camino entre los dos índices de vértices, son idénticas en los dos algoritmos. Las entradas diferentes de cero en las matrices del algoritmo de Warshall denotan la existencia de un camino entre

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6 1 5 6 4 2 4 0 0 5 6 8 7	0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 2 6 7 5 3 5 0 0 6 7 9 8	0 0 1 0 3 1 0 0 0 0 0 0 0	0 0 0 2 3 3 0 0 0 0 0 0 0	0 0 0 4 2 5 0 0 0 0 0 0 0	2 3 1 3 1 4 6 0 0 1 2 4 3	5 6 4 6 4 7 3 2 1 4 5 7 6	6 7 5 8 4 1 2 5 6 8 7 0 0	0 0 0 0 0 0 0 0 0 1 3 2 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0	7 8 6 9 5 0 0 6 7 9 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0
B	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
D	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
E	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
F	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
G	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
H	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
I	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
J	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
K	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
L	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0
M	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figura 32.7 Etapas finales del algoritmo de Floyd.

los dos vértices; para el algoritmo de Floyd estos valores proporcionan la longitud del camino más corto de los descubiertos hasta ahora. El camino más corto real puede también calcularse utilizando una versión matricial del array padre de los capítulos anteriores: poniendo en la entrada de la fila x y columna j el nombre del vértice anterior en el camino más corto desde x a j (el vértice y del bucle interno del programa anterior).

Ordenación topológica

Los grafos cíclicos aparecen en muchas de las aplicaciones en las que intervienen los grafos dirigidos. Si el grafo de la Figura 32.1 modela una cadena de producción, esto implica que la tarea A debe hacerse antes que la tarea G, ésta antes que la C, y ésta antes que la A. Pero tal situación es inconsistente: para esta y otras muchas aplicaciones hay que invocar a los grafos dirigidos con *ciclos no dirigidos* (ciclos con todas las aristas apuntando en la misma dirección). Tales grafos se denominan *grafos dirigidos acíclicos*, o en acrónimo *DAGs*¹. Los dags pueden tener numerosos ciclos, si no se tienen en cuenta las direcciones de las aristas; pero por su propia definición es evidente que no se puede llegar nunca a un ciclo siguiendo las aristas en la dirección indicada. La Figura 32.8 muestra un dag similar al grafo dirigido de la Figura 32.1, pero en el que se han eliminado algunas aristas o cambiado direcciones con el fin de suprimir ciclos. La lista de aristas de este grafo es la misma que la del grafo conexo del Capítulo 30, pero aquí, otra vez, sí tiene importancia el orden en el que se dan los vértices cuando se especifica la arista.

Los dags son realmente muy diferentes de los grafos dirigidos en general: en cierto sentido, son a la vez árbol y grafo. Es posible aprovecharse de su estruc-

¹ Acrónimo de *Directed Acyclic Graph*. (N. del T.)

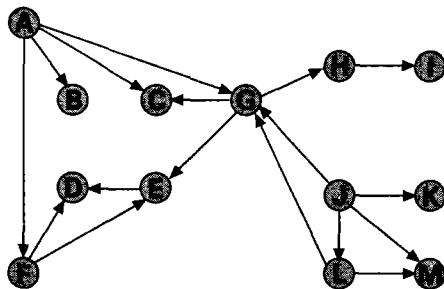


Figura 32.8 Un grafo dirigido acíclico.

tura especial cuando se los procesa. Desde el punto de vista de cualquier vértice, un dag parece un árbol: de otra forma, el bosque de búsqueda en profundidad de un dag no tiene aristas ascendentes. La Figura 32.9 proporciona el bosque de búsqueda en profundidad del dag de la Figura 32.8.

Una operación fundamental sobre los dags es el tratamiento de los vértices del grafo en un orden tal que ningún vértice se procese antes de cualquier otro que apunte hacia él. Por ejemplo, los nodos del grafo anterior deben procesarse en el siguiente orden:

J K L M A G H I F E D B C

Si las aristas se dibujaran con los vértices en estas posiciones, todas estarían dirigidas de izquierda a derecha. Como se mencionó anteriormente, esto tiene aplicaciones evidentes, por ejemplo, en los grafos que representan procesos de fabricación, porque proporciona un medio específico de trabajar de acuerdo con las restricciones representadas por los gráficos. Esta operación se denomina *ordenación topológica*, dado que implica una ordenación de los vértices del grafo.

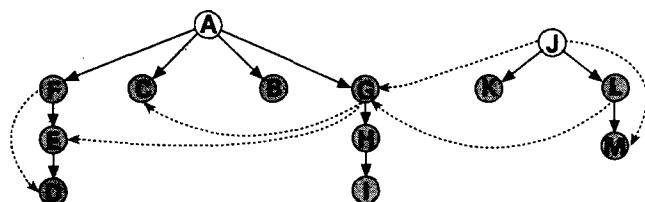


Figura 32.9 Búsqueda en profundidad en un dag.

En general, el orden de los vértices producido por una ordenación topológica no es único. Por ejemplo, el orden

A J G F K L E M B H C I D

es un orden topológico válido para el ejemplo (y existen muchos otros). En la mencionada aplicación de fabricación, esta situación aparece cuando una tarea no tiene dependencia directa o indirecta de otra, y por ello se pueden realizar en cualquier orden.

A veces es útil interpretar a la inversa a las aristas del grafo: es decir, que una arista dirigida de x a y significa que el vértice x «depende» del y . Por ejemplo, los vértices podrían representar conceptos a definir en un manual de programación (¡o en un libro de algoritmos!) con una arista desde x hacia y si la definición de x utiliza y . En este caso sería útil encontrar un orden con la propiedad de que todo término se define antes de que se utilice en otra definición. Esto corresponde a la disposición de vértices en una línea de tal forma que las aristas vayan todas de derecha a izquierda. Un *orden topológico inverso* del grafo ejemplo es:

D E F C B I H G A K M L J

Aquí la distinción no es crucial: la realización en un grafo de una ordenación topológica inversa es equivalente a llevar a cabo una ordenación topológica en el grafo que se obtiene al invertir el sentido de todas las aristas.

Pero ya se ha visto un algoritmo para la ordenación topológica inversa, ¡el procedimiento recursivo estándar de búsqueda en profundidad del Capítulo 29!

Cuando el grafo de entrada es dirigido y sin ciclos, al añadir una instrucción (por ejemplo insertar `cout << nombre(k)` antes del final) induce a `bp` a imprimir los vértices en un orden topológico inverso. Un simple razonamiento por inducción demuestra que esto funciona: se imprime el nombre de cada vértice después de haberlo hecho con los de todos los vértices a los que apunta. Cuando se cambia `visitar` de esta forma y se aplica al ejemplo, imprime los vértices en el orden topológico inverso antes dado. La impresión de los nombres de los vértices antes de terminar este procedimiento recursivo equivale exactamente a colocar el nombre del vértice en una pila de entrada, después sacarlo de la pila e imprimir la salida. No hay razón para utilizar explícitamente una pila en este caso, ya que el mecanismo de recursión proporciona una automáticamente; sin embargo, se necesitará una para el problema más difícil que se tratará a continuación.

Componentes fuertemente conexas

Si un grafo contiene un *ciclo dirigido* (si se puede volver a obtener un nodo siguiendo aristas en la dirección indicada) entonces no es un dag y no puede or-

denarse topológicamente: cualquier vértice del ciclo que se imprima primero tendrá otro vértice que apunta hacia él y que todavía no se ha impreso. Los nodos del ciclo son mutuamente accesibles en el sentido de que existe una forma de ir desde cualquier nodo a otro y regresar. Por otra parte, incluso aunque un grafo pueda ser conexo, es improbable que se pueda obtener algún nodo desde cualquier otro por medio de un camino dirigido. De hecho, los nodos se dividen ellos mismos en conjuntos denominados *componentes fuertemente conexas* que tienen la propiedad de que todos los nodos del interior de una componente son mutuamente accesibles, pero no existe ningún medio de ir desde un nodo de una de las componentes a otro de otra componente y regresar. Las componentes fuertemente conexas del grafo dirigido de la Figura 32.1 son los dos nodos sencillos B y K, el par H I, la tripleta D E F y una gran componente con seis nodos A C G J L M. Por ejemplo, el vértice A está en una componente diferente de la del vértice F porque, aunque existe un camino desde A hacia F, no lo hay para ir de F hacia A.

Las componentes fuertemente conexas de un grafo dirigido pueden encontrarse utilizando una variante de la búsqueda en profundidad, como era de esperarse. El método que se examinará fue descubierto por R. E. Tarjan en 1972. Como se basa en la búsqueda en profundidad, su tiempo de ejecución es proporcional a $V + A$, pero se trata realmente de un método ingenioso, que necesita solamente algunas modificaciones simples del procedimiento básico *visitar*, y, a pesar de ello, antes de que Tarjan lo presentara, no se conocía ningún algoritmo lineal que resolviera este problema, aunque mucha gente había trabajado en él.

La versión modificada de la búsqueda en profundidad que se va a utilizar para encontrar las componentes fuertemente conexas de un grafo es muy parecida al programa que se estudió en el Capítulo 30 para encontrar componentes biconexas. La función recursiva *visitar* dada anteriormente utiliza el mismo cálculo de *min* para encontrar el vértice más alto que sea accesible (por medio de un enlace ascendente) desde cualquier descendiente del vértice *k*, pero utiliza el valor de *min* de una forma ligeramente diferente con objeto de listar las componentes fuertemente conexas:

```
Pila pila(maxV);
int visitar(int k) // BP para encontrar componentes fuertes
{
    struct nodo *t; int m, min;
    val[k] = ++id; min = id;
    pila.meter(k);
    for (t = ady[k]; t != z; t = t->siguiente)
    {
        m = (!val[t->v]) ? visitar(t->v) : val[t->v];
        if (m < min) min = m;
    }
}
```

```

if (min == val[k])
{
    do
    {
        m = pila.sacar(); cout << m;
        val[m] = V+1;
    }
    while (m != k);
    cout << '\n';
}
return min;
}

```

Este programa coloca los nombres de los vértices en una pila en la entrada de visitar, y a continuación los saca e imprime la salida en el momento en que se acabe de explorar el último miembro de cada componente fuertemente conexa. La clave de este procedimiento es comprobar si min y $\text{val}[k]$ son iguales: en ese caso, todos los vértices que se han encontrado desde la entrada (excepto aquellos que ya se han listado) pertenecen a la misma componente fuertemente conexa que k . Como es habitual, este programa se puede modificar fácilmente para hacer un procesamiento más sofisticado que una simple escritura de las componentes.

Propiedad 32.3 *Las componentes fuertemente conexas de un grafo pueden encontrarse en tiempo lineal.*

Una demostración completa y rigurosa de que el algoritmo anterior calcula las componentes fuertemente conexas está fuera del alcance de este libro, pero se pueden presentar las ideas principales. El método se basa en dos observaciones que ya se han hecho en otros contextos. Primero, una vez que se alcanza el final de la llamada a `visitar` para un vértice, no se puede encontrar ningún vértice de la misma componente fuertemente conexa (puesto que todos los vértices que se pueden alcanzar desde ese vértice ya han sido procesados, como se expresó anteriormente en la ordenación topológica). Segundo, los enlaces ascendentes del árbol proporcionan un segundo camino desde un vértice hacia otro y reúnen conjuntamente a las componentes fuertes. Como con el algoritmo del Capítulo 30, para encontrar los puntos de articulación, se sigue la pista del ascendiente más alto accesible por medio de un enlace ascendente desde todos los descendientes de cada nodo. Ahora, si un vértice x no tiene descendientes o enlaces ascendentes en el árbol de búsqueda en profundidad, o si tiene un descendiente en dicho árbol con un enlace ascendente que apunta a x y no tiene descendientes con enlaces ascendentes que apunten más alto en el árbol, entonces él y todos sus descendientes (excepto aquellos vértices que satisfacen la misma propiedad y sus descendientes) forman una componente fuertemente conexa.

De esta forma, en el árbol de búsqueda en profundidad de la Figura 32.2, los nodos B y K cumplen la primera condición (por lo que representan por sí mismos componentes fuertemente conexas), y los nodos F (representando F E D), H (representando H I) y A (representando A G J L M C) satisfacen la segunda condición. Los miembros de la componente representada por A se encuentran eliminando B K F y sus descendientes (que aparecen en las componentes ya descubiertas). Cada descendiente y de x que no satisface esta misma propiedad tiene algún descendiente con un enlace ascendente apuntando más alto que y en el árbol. Existe un camino descendente de x a y en el árbol y además se puede encontrar un camino que vaya de y a x descendiendo desde y hacia el vértice con un enlace ascendente que sobrepase a y , continuando con el mismo procedimiento hasta alcanzar x . Un rasgo característico crucial, suplementario, es que, una vez que se ha acabado con un vértice, se le atribuye un *val* alto, de manera que se puedan ignorar los enlaces transversales que apuntan hacia él. ■

Este programa proporciona una solución sorprendentemente simple a un problema de relativa dificultad. Es una prueba cierta de las sutilezas propias de la búsqueda en grafos dirigidos, sutilezas que pueden allanarse (en este caso) por medio de un programa recursivo cuidadosamente pensado.

Ejercicios

1. Presentar la matriz de adyacencia de la clausura transitiva del dag de la Figura 32.8.
2. ¿Cuál sería el resultado de ejecutar los algoritmos de clausura transitiva en un grafo dirigido que está representado por una matriz de adyacencia?
3. Escribir un programa para determinar el número de aristas de la clausura transitiva de un grafo dirigido dado, utilizando la representación por lista de adyacencia.
4. Comparar el algoritmo de Warshall con el de la clausura transitiva derivado de la utilización de la técnica de búsqueda en profundidad descrita en el texto, pero utilizando *visitar* con la forma de matriz de adyacencia y eliminando la recursión.
5. Presentar la ordenación topológica generada por el dag de la Figura 32.8 cuando se utiliza el método sugerido, con una representación por matriz de adyacencia, pero en el que bp explora los vértices en orden inverso (desde V hacia 1) durante el examen de los vértices no vistos.
6. ¿El algoritmo del camino más corto del Capítulo 31 es válido para grafos dirigidos? Explicar por qué o dar un ejemplo en el caso de que no sirva.
7. Escribir un programa para determinar si un grafo dado es un dag o no.
8. ¿Cuántas componentes fuertemente conexas existen en un dag? ¿Y en un grafo con un ciclo dirigido de tamaño V ?
9. Utilizar los programas de los Capítulos 29 y 30 para generar grandes grafos

aleatorios dirigidos con V vértices. ¿Cuántas componentes fuertemente conexas tienen tendencia a tener tales grafos?

10. Escribir un programa que sea funcionalmente análogo a pertenencia del Capítulo 30, pero que conserve las componentes *fuertemente* conexas del grafo *dirigido* descrito por las aristas de entrada. (Este no es un problema fácil; lo más seguro es que no se podrá obtener un programa tan eficaz como pertenencia.)

Flujo de red

Los grafos dirigidos ponderados son modelos muy útiles en ciertos tipos de aplicaciones que implican flujo de productos a través de una red de interconexión. Considérese, por ejemplo, una red de oleoductos de distintos tamaños, interconectados de forma compleja, con válvulas que controlen la dirección del flujo en las derivaciones. Supóngase además que la red tiene una fuente única (como un campo de petróleo) y un único destino (como una gran refinería) al que convergen finalmente todos los conductos. ¿Qué ajuste de válvulas hará máxima la cantidad de petróleo que fluye de la fuente hacia el destino? Las interacciones complejas sobre el flujo de productos en las derivaciones hacen que la resolución de este *problema del flujo de red* no sea trivial.

El mismo esquema general puede utilizarse para describir el tráfico que fluye por las autopistas, los materiales que fluyen de las factorías, etc. Se han estudiado numerosas versiones de este problema, correspondientes a las diferentes situaciones prácticas en las que se puede aplicar, y existe evidentemente un gran interés en encontrar un algoritmo eficaz para estos problemas.

Este tipo de problema está situado como una interfaz entre la informática y el campo de la *investigación operativa*, en la que el interés se centra en la modelación matemática de sistemas complejos con objeto de ayudar en la toma de decisiones (preferentemente óptimas). El flujo de red es un ejemplo típico de un problema de investigación operativa; algunos otros se tratarán someramente en los Capítulos 42-45.

En el Capítulo 43 se estudiará la *programación lineal* que es una aproximación general a la resolución de ecuaciones matemáticas complejas que normalmente son el resultado de modelos de investigación operativa. Para problemas específicos, como el del flujo de red, existen algoritmos mejores. De hecho, se verá que la solución clásica al problema del flujo de red está muy relacionada con los algoritmos sobre grafos que se acaban de estudiar y que es bastante fácil desarrollar un programa para resolver el problema utilizando las herramientas algorítmicas que se han descubierto. Pero este problema es uno de los que todavía se están estudiando activamente: al contrario de muchos de los problemas

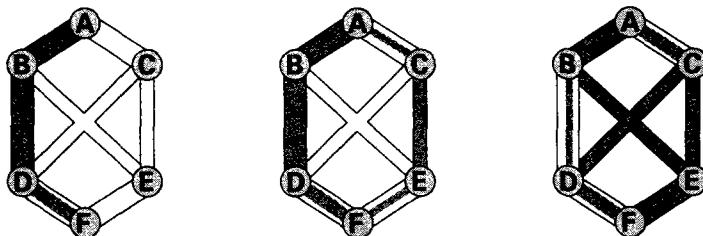


Figura 33.1 Flujo máximo en una red simple.

que se han visto, todavía no se ha encontrado la «mejor» solución, y aún se están descubriendo nuevos algoritmos.

El problema del flujo de red

Considérese la representación gráfica idealizada de la pequeña red de oleoductos que se muestra en la Figura 33.1. Las canalizaciones tienen una capacidad fija proporcional a su tamaño y el petróleo solamente puede fluir en forma descendente (de arriba abajo). Además las válvulas de cada derivación controlan la cantidad de petróleo que va en cada dirección. Sin importar la forma en la que están puestas las válvulas, el sistema alcanza un estado de equilibrio cuando la cantidad de petróleo que fluye al sistema por arriba es igual a la cantidad que fluye hacia afuera por la parte inferior (ésta es la cantidad que se desea maximizar) y cuando la cantidad de petróleo que fluye hacia cada derivación es igual a la que sale de ella. El flujo de petróleo y la capacidad de las tuberías se miden en una unidad determinada (por ejemplo, litros por segundo).

En principio no parece evidente que el estado de las válvulas pueda afectar realmente el flujo total máximo: la Figura 33.1 muestra que sí es así. Se supone que la válvula que controla la tubería AB está abierta y que tanto AB como BD están llenas y DF casi llena, como se muestra en el diagrama a la izquierda de la figura. A continuación se supone que se abre la tubería AC y se sitúa la válvula C de forma que cierre CD y abra CE (quizás el operario de la válvula D ha informado al de la válvula C que no puede manipular por más tiempo su válvula dada la carga que viene de B). El flujo resultante se muestra en el diagrama central de la figura: las tuberías BD y CE están llenas. Ahora se podría aumentar el flujo un poco dejando pasar el suficiente petróleo por el camino ACDF para llenar la tubería DF, pero existe una mejor solución, como se muestra en el tercer diagrama. Cambiando la válvula B para que redirija el flujo suficiente

para llenar BE, se aumenta el caudal de la tubería DF para permitir que la válvula C llene completamente la tubería CD. El flujo total que entra y sale de la red se ha incrementado por medio de una adecuada regulación de las válvulas.

El objetivo es desarrollar un algoritmo que pueda encontrar el reglaje de válvulas «adecuado» para cualquier red. Además, se desea estar seguros de que ningún otro reglaje dará un flujo mayor.

Esta situación evidentemente puede modelarse por medio de un grafo dirigido, y así los programas que se han estudiado pueden aplicarse aquí. Se define una *red* como un grafo dirigido ponderado con dos vértices principales: uno, que no tiene aristas que apunten a él (*la fuente*), y otro que no tiene aristas que apunten hacia afuera de él (*el pozo*). Los pesos de las aristas, que se supone que no son negativos, se denominan *capacidades de las aristas*. Ahora, un *flujo* se define como un conjunto de pesos en las aristas tal que el flujo en cada una de ellas es igual o menor que la capacidad, y el flujo que entra en cada vértice es igual al que sale de él. El *valor del flujo de red* consiste en encontrar un flujo de valor máximo para una red dada.

Obviamente se pueden representar las redes por medio de la matriz de adyacencia o de las listas de adyacencia que se utilizaron para los grafos en capítulos anteriores. En lugar de un peso único, ahora se asocian con cada arista dos pesos, la *talla* y el *flujo*, que se pueden representar como dos campos en los nodos de las listas de adyacencia, como dos matrices en la representación por matriz de adyacencia o como dos campos de un registro único en ambas representaciones. Aunque las redes son grafos dirigidos, los algoritmos que se examinarán necesitan recorrer aristas en la dirección «equivocada», por lo que se utiliza una representación de grafo no dirigido: si existe una arista desde x hacia y cuyo tamaño es t y su flujo f , también se tiene una arista desde y hacia x con tamaño $-t$ y flujo $-f$. En una representación por lista de adyacencia es necesario mantener enlaces que conecten las dos listas de nodos que representan a cada arista, de forma que una modificación del flujo en una pueda modificar el de la otra.

Método de Ford-Fulkerson

La aproximación clásica al problema del flujo de red fue desarrollada por L. R. Ford y D. R. Fulkerson en 1962, quienes proporcionaron un método para mejorar cualquier flujo admisible (excepto, por supuesto, el máximo). Comenzando con un flujo cero, se aplica el método de forma iterativa; cuanto más se aplique, más aumento de flujo producirá, y si no se puede aplicar más es que se ha encontrado el flujo máximo. De hecho, el flujo de la Figura 33.1 se ha desarrollado utilizando este método, y a continuación se vuelve a examinar por medio de la representación del grafo que se muestra en la Figura 33.2.

Para simplificar se omiten las flechas, dado que todas apuntan hacia abajo.

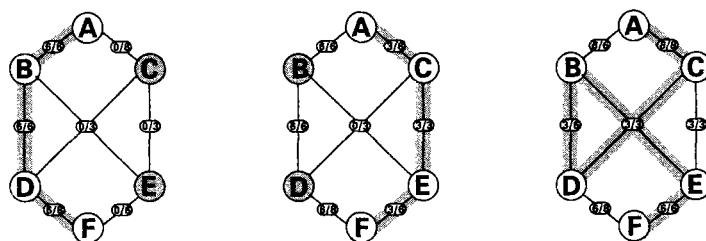


Figura 33.2 Búsqueda del flujo máximo de una red.

Los métodos que se van a considerar no se limitan a los grafos que se pueden dibujar con todas las aristas apuntando en una sola dirección. Se utilizan tales grafos porque proporcionan una buena intuición para poder comprender el flujo de red en términos de líquidos que fluyen por canalizaciones.

Considerando cualquier camino dirigido (hacia abajo) a través de la red (de la fuente al pozo), evidentemente el flujo se puede incrementar por lo menos en la más pequeña cantidad de capacidad no utilizada en cualquier arista del camino, incrementando el flujo en todas las aristas del camino en esa cantidad. En el diagrama de la izquierda de la Figura 33.2 esta regla se aplica al camino ABDF; después en el diagrama central se aplica a lo largo del camino ACEF.

Como se mencionó anteriormente, se podría aplicar la misma regla al camino ACDF, lo que produciría una situación en la que todos los caminos dirigidos que atraviesan la red tienen al menos una arista con capacidad completa. Pero existe otra forma de incrementar el flujo: se pueden considerar caminos arbitrarios a través de la red que contienen a las aristas que apuntan en la «dirección equivocada» (desde el pozo hacia la fuente a lo largo del camino). El flujo puede incrementarse a lo largo de tal camino *incrementando* el de las aristas que van de la fuente al pozo y *reduciendo* en la misma cantidad el de las aristas que van desde el pozo a la fuente. En el ejemplo, el flujo a través de la red se puede incrementar en tres unidades a lo largo del camino ACDBEF, como se muestra en el tercer diagrama de la Figura 33.2. Como se describió anteriormente, esto corresponde a añadir tres unidades al flujo a través de AC y CD, y desviar tres unidades por la válvula B desde BD hacia BE y EF. No hay pérdida de flujo en DF porque tres de las unidades que se han utilizado venían antes de BD y ahora lo hacen de CD.

Para simplificar la terminología, se denominará a las aristas que fluyen de la fuente al pozo, a lo largo de un camino particular, aristas *de ida*, y a las aristas que fluyen desde el pozo hacia la fuente, aristas *de vuelta*. Obsérvese que la cantidad por la que se puede aumentar el flujo está limitada por el mínimo de las capacidades sin utilizar de las aristas de ida y el mínimo de los flujos de las aristas de vuelta. De otra forma, en el nuevo flujo, al menos una de las aristas de ida a lo largo del camino se llena o al menos una de las aristas de vuelta a lo

largo del camino se vacía. Además, no se puede incrementar el flujo de un camino cualquiera que contenga una arista de ida llena o una arista de vuelta vacía.

El párrafo anterior proporciona un método para incrementar el flujo de cualquier red, *a condición* de que se pueda encontrar un camino que no tenga aristas de ida llenas o aristas de vuelta vacías. El punto crucial del método de Ford-Fulkerson está en el hecho de que si no se puede encontrar un camino de este tipo entonces el flujo es máximo.

Propiedad 33.1 *Si todos los caminos desde la fuente hacia el pozo de una red tienen una arista de ida llena o una arista de vuelta vacía, el flujo es máximo.*

Para demostrar este hecho, primero se recorre el grafo y se identifica la primera de las aristas de ida llena o de vuelta vacía de cada camino. Este conjunto de aristas *corta* el grafo en dos partes. (En el ejemplo, las aristas AB, CD y CE hacen este corte.) Para cualquier corte de la red en dos partes, se puede medir el flujo «a través» del corte: el total del flujo en las aristas que va desde la fuente al pozo.

En general, las aristas pueden atravesar el corte en los dos sentidos: para obtener el flujo a través del corte se debe sustraer el total del flujo en las aristas que van en sentido contrario. En el ejemplo, el corte tiene el valor 12, que es igual al total del flujo de la red. Cada vez que el flujo del corte es igual al flujo total se sabe que no solamente el flujo es máximo, sino que también el corte es mínimo (es decir, que cualquier otro corte tiene al menos un «flujo de cruce» mayor). Éste es el denominado *teorema del flujo máximo y corte mínimo*: el flujo no podría ser más grande (en caso contrario el corte podría ser más grande también), y no existen cortes más pequeños (en caso contrario el flujo podría ser más pequeño también). Se omiten los detalles de la demostración. ■

Búsqueda en red

El método de Ford-Fulkerson descrito anteriormente puede resumirse de la siguiente forma: «comenzar por todas partes con un flujo cero e incrementarlo a lo largo de cualquier camino desde la fuente al pozo que no contenga aristas de ida llenas o de vuelta vacías, continuando hasta que no exista en la red un camino de este tipo». Pero esto no es un *algoritmo* en el sentido habitual, porque como no se ha especificado el método de búsqueda de caminos se podría utilizar cualquiera de todos ellos. Por ejemplo, se podría basar el método en la intuición de que cuanto mayor es el camino, más se llenará la red, y por ello se prefieren los caminos más largos. Pero el ejemplo (clásico) que se muestra en la Figura 33.3 demuestra que se debe ser prudente.

En esta red, si el primer camino elegido es ABCD, entonces el flujo se incrementa solamente en una unidad. Entonces el segundo camino elegido podría

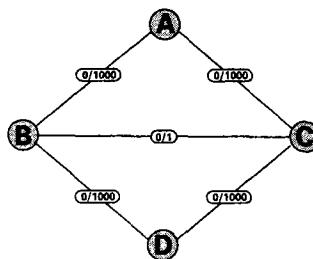


Figura 33.3 Una red que puede necesitar un gran número de iteraciones.

ser ACDB, incrementando otra vez el flujo en una unidad y dejando una situación idéntica a la inicial, excepto que el flujo en las aristas exteriores se ha incrementado en una unidad. Cualquier algoritmo que elija estos dos caminos (por ejemplo aquel que prefiera los caminos largos) continuará con esta estrategia, necesitándose mil parejas de iteraciones antes de encontrar el flujo máximo. Si las cantidades en los costados son iguales a mil millones, entonces harían falta dos mil millones de iteraciones. Por supuesto, esta situación no es de desear porque los caminos ABC y ADC dan el flujo máximo exactamente en dos etapas. Para que el algoritmo sea útil, se debe evitar que su tiempo de ejecución dependa de la magnitud de las capacidades. Por fortuna este problema se elimina fácilmente:

Propiedad 33.2 *Si en el método de Ford-Fulkerson se utiliza el camino más corto que existe entre la fuente y el pozo, entonces el número de caminos que se utilizan antes de encontrar el flujo máximo de una red de V vértices y A aristas debe ser inferior a VA .*

Esta afirmación fue demostrada por Edmonds y Karp en 1972. Los detalles de la misma están fuera del alcance de este libro.■

En otras palabras, una buena idea es simplemente utilizar una versión convenientemente modificada de la búsqueda en anchura para encontrar el camino que se desea. La cota mayorante proporcionada en la propiedad 33.2 corresponde al peor caso: una red tipo debería necesitar muchos menos pasos.

Con el método de recorrido de grafos del Capítulo 31, se puede implementar otro método propuesto por Edmonds y Karp: *encontrar el camino que atraviesa la red que incrementa el flujo en la mayor cantidad posible*. Esto se puede realizar simplemente utilizando una variable para prioridad (cuyo valor se establecerá adecuadamente) en los métodos de «búsqueda en primera prioridad» de lista de adyacencias o matriz de adyacencia, expuestos en el Capítulo 31. Para la representación por matriz, las instrucciones siguientes calculan la prioridad,

y el código correspondiente es similar a la representación por lista de adyacencias:

```
prioridad = -flujo[k][t];
if (talla[k][t] > 0) prioridad += talla[k][t];
if (prioridad > val[k]) prioridad = val[k];
```

Como se desea seleccionar el nodo con el valor de prioridad *más alto* se debe, o bien reorientar los mecanismos de cola de prioridad de esos programas para devolver el máximo en lugar del mínimo, o bien utilizarlos como prioridad con el complemento con respecto a algún gran entero (y el proceso inverso cuando se elimine el valor). También se modifica el procedimiento de búsqueda en primera prioridad para introducir a la fuente y el pozo como argumentos y a continuación comenzar cada búsqueda en la fuente y terminarla cuando se ha encontrado un camino hacia el pozo (devolviendo 1 si se ha encontrado un camino o 0 si no existe ninguno). Si no existe camino, el árbol de búsqueda parcial define un corte mínimo de la red; en caso contrario se puede aumentar el flujo. Por último, el *val* de la fuente debe establecerse en *maxint* antes de que comience la búsqueda, para indicar que la fuente puede aceptar cualquier cantidad de flujo (aunque esto se restringe inmediatamente por la capacidad total de todas las canalizaciones que salen directamente de la fuente).

Con la búsqueda en primera prioridad implementada como se describe en el párrafo anterior, encontrar el flujo máximo es realmente muy simple. El programa siguiente supone una representación por matriz de adyacencia para la red:

```
for (;;)
{
    if (buscar(1,V)) break;
    y = V; x = padre[V];
    while ( x != 0 )
    {
        flujo[x][y] = flujo[x][y]+val[V];
        flujo[y][x] = -flujo[x][y];
        y = x; x= padre[y];
    }
}
```

Mientras que *buscar* pueda encontrar un camino que aumente el flujo (en la cantidad máxima), se sigue el camino «marcha atrás» (utilizando el array *padre* construido por *buscar*) y se aumenta el flujo de la forma indicada. Si *V* permanece no visto después de algunas llamadas a *buscar*, entonces se ha encontrado un corte mínimo y el algoritmo finaliza.

Como se ha visto, el algoritmo comienza por aumentar el flujo a lo largo del

camino ABDF, después a lo largo de ACEF y después a lo largo de ACDBEF. El método no elige a ACDF como tercer camino, dado que incrementaría el flujo solamente en una unidad, no en tres como es posible con el camino más largo. Es de destacar que el método del «primer camino más corto» de búsqueda en anchura de la propiedad 33.2 haría esta elección.

Aunque el algoritmo es fácil de implementar y da buenos resultados al trabajar con las redes que se encuentran en la práctica, su análisis es muy complejo. En principio, como habitualmente, buscar necesita V^2 pasos en el peor caso; de forma alternativa se podrían utilizar listas de adyacencia a ejecutar en tiempo proporcional a $(A + V)\log V$ por iteraciones, aunque el algoritmo sea verdaderamente más rápido que éste, dado que se detiene cuando alcanza el pozo. Pero ¿cuántas iteraciones se necesitan?

Propiedad 33.3 *Si se utiliza en el método de Ford-Fulkerson el camino de la fuente al pozo que incremente el flujo en la mayor cantidad posible, el número de caminos utilizados antes de encontrar el flujo máximo de una red es inferior a $1 + \log_{M/M-f^*}$, donde f^* es el coste del flujo y M el número máximo de aristas en un corte de la red.*

Una vez más, la demostración de esta propiedad, hecha en primer lugar por Edmonds y Karp, supera con creces el marco de este libro. Esta cantidad es verdaderamente complicada de calcular, pero es improbable que sea grande en las redes reales. ■

Se menciona aquí esta propiedad no para indicar lo grande que es el tiempo que lleva el algoritmo en una red real, sino por la complejidad del análisis. De hecho, este problema se ha estudiado ampliamente y se han desarrollado complejos algoritmos con cotas superiores del peor caso muy inferiores. Sin embargo, el algoritmo de Edmonds-Karp, implementado de la forma anterior, puede que encuentre dificultades en las redes que aparecen en las aplicaciones prácticas. La Figura 33.4 muestra el algoritmo operando en una red muy grande.

El problema del flujo de red puede generalizarse de varias formas, y se han estudiado con mucho detalle un gran número de variantes, que son importantes en las aplicaciones reales. Por ejemplo, el *problema de los flujos generalizados* implica la introducción en la red de múltiples fuentes, pozos y tipos de productos. Esto hace que el problema sea mucho más difícil y necesita algoritmos mucho más avanzados que los que se han considerado aquí: por ejemplo, no se conocen analogías a los teoremas del flujo máximo y mínimo corte para incluir en el caso general. Otras ampliaciones del problema del flujo de red incluyen la aplicación de restricciones en los vértices (fácilmente tratadas al introducir aristas artificiales para tratar esas restricciones), autorización de aristas no dirigidas (también fácilmente tratadas al reemplazar las aristas no dirigidas por pares de aristas dirigidas) e introducción de minorantes en los flujos de las aristas (no tan fáciles de tratar). Si se hace una hipótesis realista, según la cual las tuberías tienen costes asociados además de capacidades, entonces se obtiene el problema

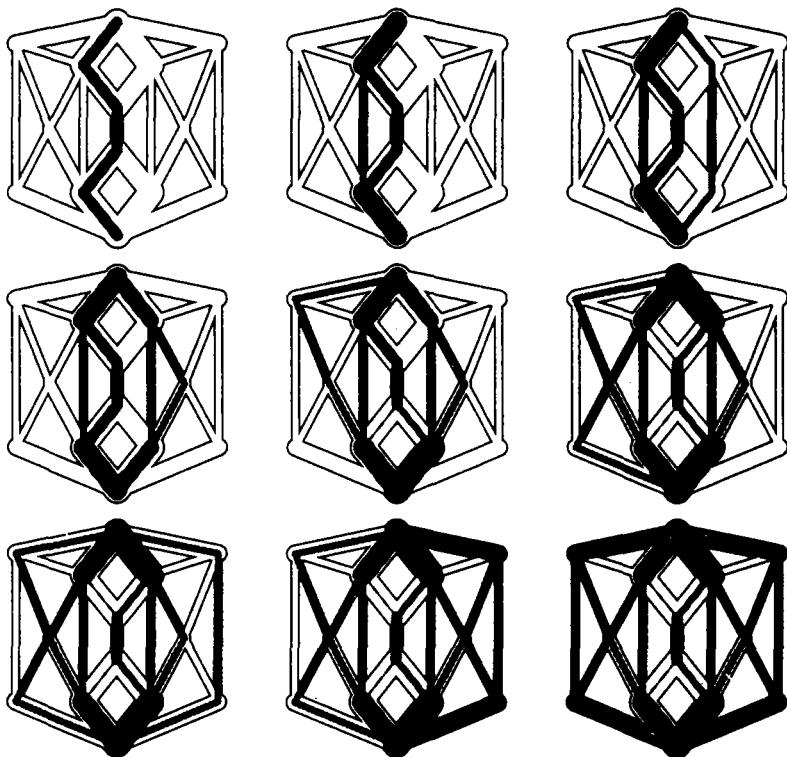


Figura 33.4 Búsqueda del flujo máximo en una red muy grande.

del flujo de *coste mínimo*, uno de los problemas más difíciles de la investigación operativa (sobre operaciones).

Ejercicios

1. Encontrar un algoritmo para resolver el problema del flujo de red para el caso en que la red forma un árbol si se elimina la fuente.
2. ¿Qué caminos se siguen en el algoritmo propuesto en la propiedad 33.3 al encontrar el flujo máximo de una red, obtenido al añadir aristas de peso 3 desde B a C y de E a D?
3. Dibujar los árboles de búsqueda en primera prioridad calculados en cada llamada a buscar para el ejemplo presentado en el texto.
4. Obtener el contenido de la matriz flujo después de cada llamada a buscar para el ejemplo presentado en el texto.

5. ¿Es cierto que ningún algoritmo puede encontrar el flujo máximo sin examinar cada arista de la red?
6. ¿Qué sucede en el método de Ford-Fulkerson cuando la red tiene un ciclo dirigido?
7. Obtener una versión simplificada de la cota de Edmonds-Karp para el caso de que todas las capacidades estén en $O(1)$.
8. Obtener un contraejemplo que muestre por qué la búsqueda en profundidad no es apropiada para el problema del flujo de red.
9. Implementar la solución de la búsqueda en anchura del problema del flujo de red, utilizando una búsqueda en primera prioridad con una representación por listas de adyacencia.
10. Escribir un programa para encontrar los flujos máximos en redes aleatorias con V nodos y aproximadamente $10V$ aristas. ¿Cuántas iteraciones se hacen para $V= 25, 50, 100$?

Concordancia

En este capítulo se examinará el problema del «emparejamiento» de objetos en una estructura de grafo, o de su unión según las relaciones de preferencia que posiblemente estarán en conflicto.

Por ejemplo, en los Estados Unidos se ha establecido un sistema bastante complejo para la asignación de plazas de internos residentes en hospitales a los estudiantes de medicina que acaban de obtener la licenciatura. Cada estudiante solicita varios hospitales en orden de preferencia, y cada hospital establece una clasificación de estudiantes, también en orden de preferencia. El problema consiste en asignar las plazas de forma equitativa, respetando todas las preferencias establecidas. Se necesita un algoritmo sofisticado, dado que varios hospitales quieren elegir a los mejores estudiantes, y muchos estudiantes pretenden obtener las mejores plazas de hospital. Incluso no está claro que cada plaza pueda ser ocupada por uno de los estudiantes que ha solicitado el hospital o que cada estudiante pueda obtener una de las plazas que ha pedido, o que se respete el orden de preferencia. Esto ocurre frecuentemente; de hecho, después de que el algoritmo haya hecho lo mejor de que es capaz, existen cambios de última hora entre estudiantes y hospitales, para completar así el proceso.

Este ejemplo es un caso especial de un problema fundamental de grafos que ha sido ampliamente estudiado. Dado un grafo, una *concordancia* es un subconjunto de aristas en las que ningún vértice aparece más de una vez. Esto es, cada vértice alcanzado por una de las aristas de la concordancia está emparejado con el otro vértice de esa arista, pero algunos vértices pueden permanecer aislados. Aunque se insista en que una concordancia cubre el mayor número de vértices posible, en el sentido de que ninguna de las aristas de la concordancia debe conectar vértices sin concordancia, cada forma diferente de elegir las aristas puede dar lugar a números diferentes de vértices sobrantes (que no tienen concordancia).

Es de particular interés la *concordancia máxima*, que contiene el mayor número de aristas posible o, de forma equivalente, minimiza el número de vértices sin concordancia. Lo mejor que se puede esperar es que se encuentre un con-

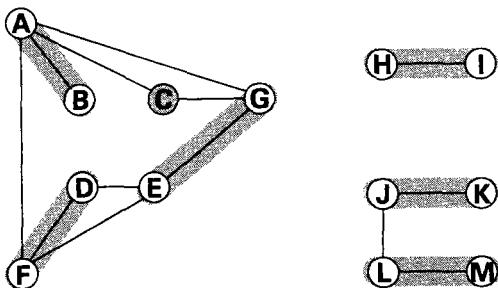


Figura 34.1 Una concordancia máxima (arista sombreadas).

junto de aristas en el que cada vértice aparezca exactamente una vez (una concordancia de este tipo en un grafo con $2V$ vértices deberá tener V aristas), pero no siempre es posible conseguir esto.

La Figura 34.1 muestra una concordancia máxima (las aristas sombreadas) para el grafo del ejemplo. Con 13 vértices no se puede hacer nada mejor que una concordancia con seis aristas. Pero los sencillos algoritmos de búsqueda de concordancias tendrán dificultades incluso en esta situación. Por ejemplo, un método podría ser intentar elegir las aristas para la concordancia tal como aparecen en la búsqueda en profundidad (véase la Figura 29.7). Para el ejemplo de la Figura 34.1 se obtendrían las cinco aristas AF EG HI JK LM, y por lo tanto no se tendría una concordancia máxima. Además, como se acaba de mencionar, no es fácil decir cuántas aristas hay en una concordancia máxima de un grafo dado. Por ejemplo, se observa que no existe concordancia de tres aristas para el subgrafo constituido por los seis vértices de A a F y las aristas que los conectan. Mientras que a menudo es muy simple obtener una gran concordancia en un gran grafo (por ejemplo, no es difícil encontrar una concordancia máxima para el grafo «laberinto» del Capítulo 29), el desarrollo de un algoritmo para encontrar la concordancia máxima de un grafo cualquiera es una tarea complicada, como lo muestran contraejemplos tales como éstos.

En el problema de la concordancia de los estudiantes de medicina descrito anteriormente, los estudiantes y los hospitales se corresponden con los nodos de un grafo; sus preferencias son las aristas. Si se asignan valores a sus preferencias (quizás con la ayuda de la tradicional escala de 1-10), entonces se tiene el problema de la *concordancia ponderada*: dado un grafo ponderado, encontrar un conjunto de aristas en el que ningún vértice aparece más de una vez y tal que la suma de los pesos de las aristas del conjunto sea máxima. Más adelante se podrá ver otra alternativa, en la que se respeta el orden de preferencia pero que no necesita (quizás arbitrariamente) que se asignen valores.

El problema de la concordancia ha atraído mucho la atención de los matemáticos dada su naturaleza intuitiva y su amplio espectro de aplicación. Su so-

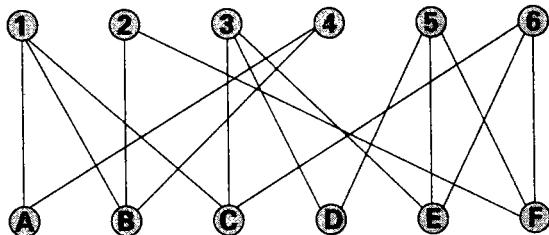


Figura 34.2 Un grafo bipartido.

lución en el caso general hace intervenir a la matemática combinatoria más bella e intrincada, lo que está fuera del alcance de este libro. Aquí se intenta proporcionar al lector los elementos para apreciar el problema, considerando los casos particulares mientras que al mismo tiempo se desarrollan algunos algoritmos de utilidad.

Grafos bipartidos

El ejemplo antes mencionado, el de la concordancia entre estudiantes de medicina y residencias, es realmente representativo de un gran número de otras aplicaciones de la concordancia. Por ejemplo, se podría concordar a los hombres y a las mujeres de un servicio matrimonial, a los trabajadores en busca de empleo y a las ofertas disponibles, a los cursos y las horas de clase, o a los diputados y los comités parlamentarios. Los grafos que aparecen en estos casos se denominan *grafos bipartidos*, que se definen como los grafos en los que todas las aristas enlazan dos conjuntos de nodos. Esto es, los nodos se dividen en dos conjuntos y ninguna arista conecta dos nodos del mismo conjunto. (Evidentemente no se desea «concordar» a un trabajador en busca de empleo con otro de su misma clase o a un comité parlamentario con otro.) Un ejemplo de grafo bipartido es el que se muestra en la Figura 34.2. El lector podría entretenese en buscar la concordancia máxima de este grafo.

En la representación por matriz de adyacencia de los grafos bipartidos se puede obtener una ganancia obvia al incluir solamente filas en un conjunto y solamente columnas en el otro. La representación por listas de adyacencia no parece sugerir ventajas por sí misma, excepto el nombrar cuidadosamente los vértices porque así es fácil decir a qué conjunto de vértices pertenece cada uno.

En los ejemplos se utilizan letras para los nodos de un conjunto y números para los nodos de otro. El problema de la concordancia máxima para los grafos bipartidos se puede expresar simplemente con esta representación: «encontrar el mayor subconjunto de un conjunto de pares de letra-número que tenga la propiedad que ningún par tenga la misma letra o número». Encontrar la con-

cordancia máxima del grafo bipartido de la Figura 34.2 corresponde a resolver este acertijo en los pares E5 A2 A1 C1 B4 C3 D3 B2 A4 D5 E3 B1.

Es un ejercicio interesante intentar encontrar una solución directa al problema de la concordancia en grafos bipartidos. En primera instancia el problema parece simple, pero las dificultades aparecen rápidamente. Existen, sin ninguna duda, demasiadas parejas posibles como para ensayar con todas ellas: la solución al problema debe ser lo suficientemente inteligente como para intentar solamente con un pequeño grupo entre todas las formas posibles de concordar los vértices.

La solución que se examinará aquí es una de las indirectas: para resolver un caso particular del problema de la concordancia, se construirá un modelo del problema de flujo de red, se utiliza el algoritmo del capítulo anterior y después se utiliza la solución del problema del flujo de red para resolver el problema de la concordancia. Esto es, se *reduce* el problema de la concordancia al problema del flujo de red. La reducción es un método de algoritmos bastante parecido a la utilización de rutinas de biblioteca por un programador de sistemas. Ésta es una técnica de fundamental importancia en la teoría de los algoritmos combinatorios avanzados (ver el Capítulo 40). Por el momento, la reducción proporcionará una solución eficaz al problema de la concordancia en los grafos bipartidos.

La construcción es inmediata: dado un elemento de la concordancia bipartida se construye un elemento del flujo de red creando un vértice fuente cuyas aristas apuntan a todos los miembros de uno de los conjuntos del grafo bipartido, y a continuación se hace que todas las aristas del grafo bipartido apunten desde este conjunto hacia el otro, y después se añade un vértice pozo al que apuntan todos los miembros del otro conjunto. Todas las aristas del grafo resultante tienen una capacidad igual a uno.

La Figura 34.3 muestra lo que sucede cuando se construye el problema del flujo de red a partir del grafo bipartido de la Figura 34.2, utilizando después el algoritmo del flujo de red del capítulo anterior. Obsérvese que la propiedad de bipartidismo del grafo, la dirección del flujo y el hecho de que todas las capacidades sean iguales a uno obligan a que cada camino a través del flujo corresponda a una arista de la concordancia: en el ejemplo, los caminos que se encuentran en las cuatro primeras etapas corresponden a la concordancia parcial A1 B2 C3 D5. Cada vez que el algoritmo del flujo de red llama a buscar o bien encuentra un camino que aumenta el flujo o finaliza.

En el quinto paso todos los caminos de la red están llenos, y el algoritmo debe utilizar las aristas de regreso. El camino que se encuentra en esta etapa, 4B2F, incrementa claramente el flujo de la red, como se vio en el capítulo anterior. En el contexto presente se puede representar el camino como un conjunto de instrucciones para crear una nueva concordancia parcial (con una arista de más) a partir del actual. Esta construcción deriva de forma natural del recorrido del camino en orden: «4A» significa añadir A4 a la concordancia, «B2» significa quitar B2 y «2F» significa añadir F2. Así, después de procesado este camino, se tiene la concordancia A1 B4 D3 D5 E6 F2; de forma equivalente, el

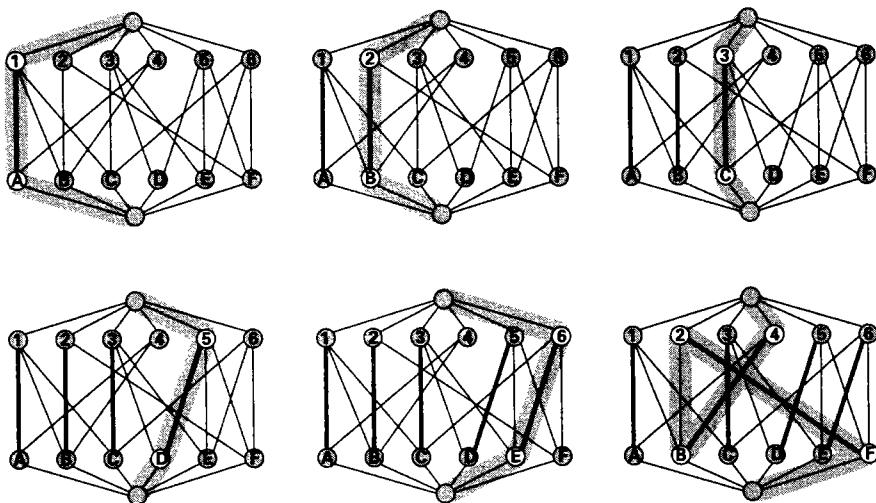


Figura 34.3 Utilización de un flujo de red para encontrar una concordancia máxima en un grafo bipartido.

flujo de red se obtiene por todas las tuberías de las aristas que conectan a estos nodos. El algoritmo finaliza haciendo la concordancia F_6 ; todas las tuberías que dejan la fuente y entran en el pozo están llenas, por lo que se tiene una concordancia máxima.

La prueba de que la concordancia consiste exactamente en aquellas aristas puestas a plena capacidad por el algoritmo de flujo máximo es directa. Primero, el flujo de red siempre obtiene una concordancia válida: como cada vértice tiene una arista de capacidad uno que o bien regresa (desde el pozo) o sale (hacia la fuente), al menos una unidad de flujo puede ir a través de cada vértice, lo que implica que cada vértice puede estar incluido al menos una vez en la concordancia. Segundo, ninguna concordancia puede tener más aristas, dado que toda concordancia de este tipo conduciría directamente a un mejor flujo producido por el algoritmo de flujo máximo.

Así, para calcular el acoplamiento máximo para un grafo bipartido se presentará simplemente el grafo en un formato aceptable como entrada del algoritmo del flujo de red del capítulo anterior. Por supuesto, los grafos presentados al algoritmo del flujo de red son mucho más simples en este caso que los grafos generales para los que se ha concebido el algoritmo y a su vez el algoritmo es algo más eficaz en este caso.

Propiedad 34.1 Una concordancia máxima en un grafo bipartido puede encontrarse en $O(V^3)$ pasos si el grafo es denso o en $O(V(A+V)\log V)$ pasos si el grafo es disperso.

La construcción asegura que cada llamada a buscar añade una arista a la concordancia, por lo que se sabe que existen al menos $V/2$ llamadas a buscar durante la ejecución del algoritmo. Por eso, el tiempo empleado es proporcional a un factor V veces mayor que el tiempo empleado en una búsqueda sencilla como la presentada en el Capítulo 31.■

Problema del matrimonio estable

El ejemplo proporcionado al comienzo de este capítulo, respecto a los estudiantes de medicina y los hospitales, evidentemente es tomado muy en serio por los participantes. Pero el método que se examinará para hacer la concordancia puede comprenderse mejor en el cuadro de un modelo más humorístico de la situación. Se supone que se tienen N hombres y N mujeres que han expresado sus preferencias mutuas (cada hombre debe decir exactamente lo que siente por cada una de las mujeres y viceversa). El problema consiste en encontrar un conjunto de N matrimonios respetando las preferencias de cada uno.

¿Cómo se deben expresar las preferencias? Un método sería utilizar una escala de «1 a 10»: cada persona asigna un valor absoluto a ciertos miembros del otro sexo. Esto transforma el problema del matrimonio al mismo que el de una concordancia ponderada, de resolución relativamente difícil. Además, la utilización de escalas absolutas por sí mismas puede llevar a incoherencias porque la escala de las personas será inconsistente (el 10 de una mujer puede ser el 7 de otra). Una forma más natural de expresar las preferencias es dejar que cada persona liste en orden de preferencia a todas las personas del sexo opuesto. La Figura 34.4 muestra un conjunto de las listas de preferencias que se podría encontrar en un grupo de cinco mujeres y cinco hombres. Como es habitual (y para proteger la inocencia!) se supondrá que se ha utilizado una técnica de dispersión o cualquier otro método para traducir los nombres reales a números de una sola cifra para las mujeres y a palabras de una sola letra para los hombres.

Evidentemente, estas preferencias entran a menudo en conflicto: por ejemplo, las dos listas A y C sitúan a 2 como su primera elección, y nadie parece querer mucho a 4 (pero alguien deberá finalmente casarse con ella). El problema consiste en incitar a cada uno a echarse novia respetando lo más posible sus preferencias y después llevar a cabo N matrimonios con un gran ceremonial. En la búsqueda de una solución se debe suponer que cada individuo prometido con alguien situado en la parte inferior de sus preferencias estará decepcionado, y siempre preferirá a alguien mejor situado en su propia lista de preferencias. Un conjunto de matrimonios se denomina *inestable* si dos personas que no están casadas se prefieren mutuamente al resto del conjunto. Por ejemplo, la asignación A1 B3 C2 D4 E5 es inestable porque A prefiere a 2 más que a 1, y 2

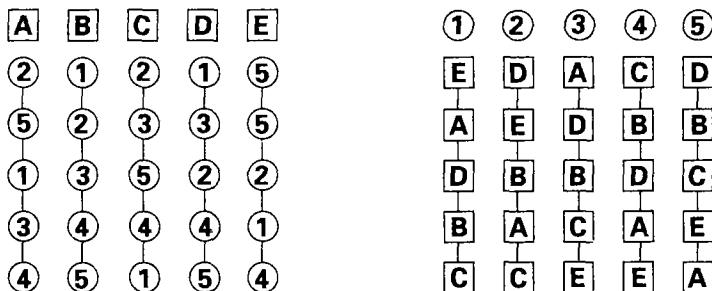


Figura 34.4 Listas de preferencia para el problema de los matrimonios estables.

prefiere a A más que a C. Así, actuando de acuerdo con sus preferencias, A debería abandonar a 2 por 1 y 2 debería dejar a C por A (no dejando a 1 y a C otra opción que unirse).

La búsqueda de una configuración estable parece una impresionante tarea porque existen muchos acuerdos. Incluso no es nada simple determinar si una configuración dada es estable, como puede comprobar el lector buscando (antes de leer el párrafo siguiente) la pareja inestable del ejemplo anterior una vez que se hayan formado las parejas A2 y C1. En general, aunque existan muchos acuerdos estables para un conjunto dado de listas de preferencias, es suficiente con encontrar uno. (Descubrir *todos* los acuerdos estables es un problema bastante difícil.)

Una posibilidad para buscar una configuración estable sería deshacer los matrimonios inestables uno por uno. Esta solución es lenta por el tiempo que se necesita para determinar la estabilidad, y además ¡puede que no tenga nunca fin! Así, después de hacer concordar a A2 y C1 en el ejemplo anterior, B y 2 forman una pareja estable, lo que conduce a la configuración A3 B2 C1 D4 E5, en la que B y 1 forman una pareja inestable, lo que conduce a la configuración A3 B1 C2 D4 E5. Finalmente, A y 1 forman una pareja inestable y se vuelve a la configuración inicial. Un algoritmo que intente resolver el problema de los matrimonios estables deshaciendo las parejas inestables una tras otra tiene muchas posibilidades de caer en un bucle infinito de este tipo.

Aquí se estudiará el caso contrario, es decir, un algoritmo que intente construir parejas estables utilizando sistemáticamente un método basado en lo que podría suceder en una versión «real» un poco idealizada del problema. La idea consiste en pedir a cada hombre, uno a uno, que haga de pretendiente y se declare a una mujer. Evidentemente lo primero que debe hacer el pretendiente en su «pedida» es pedir la mano de la primera mujer de su lista. Si ésta ya está emparejada con el hombre que ella prefiere, el pretendiente no puede hacer otra cosa que intentar con la segunda de su lista y continuar hasta encontrar una mujer sin novio o que le prefiera a él a su novio actual. En el primer caso los

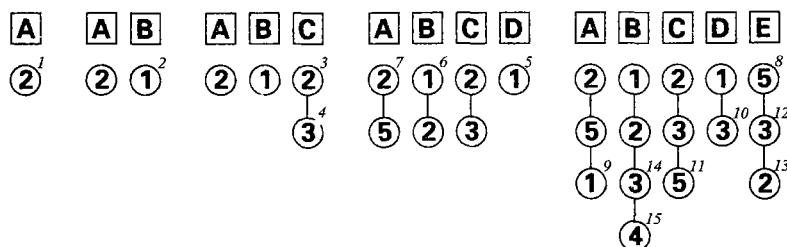


Figura 34.5 Solución del problema de los matrimonios estables.

dos se emparejan y el siguiente candidato hombre se convierte en pretendiente. En el segundo, ella rompe sus relaciones y toma al primer pretendiente (al que prefiere) como novio. Su ex-novio no puede hacer más que convertirse en pretendiente, volviendo a coger su petición donde la había dejado en la lista y acabar por encontrar una nueva elegida, lo que posiblemente provoque que una nueva pareja se deshaga, y se continúa así, de rupturas a noviazgos, hasta que un pretendiente encuentre a una mujer sin novio.

Este método parece ser un modelo de ciertas novelas del siglo XIX, pero es preciso hacer un análisis minucioso para convencerse de que genera un buen conjunto de matrimonios estables. La Figura 34.5 muestra la serie de acontecimientos en las primeras etapas del proceso aplicado al ejemplo. A pide la mano de 2 (su primera elección) que le es concedida; B pide a continuación la de 1 (su primera elección) que le es también concedida; después C pide la mano de 2, lo que se le niega y pide la de 3, la que se le concede, como muestra el tercer diagrama.

Cada diagrama presenta la serie de acontecimientos que acompañan a la llegada de cada nuevo pretendiente. Cada línea proporciona el elemento de la lista de preferencia «utilizado» por el pretendiente correspondiente, y cada enlace está etiquetado con un entero que indica el momento en el que se utiliza cada elemento por el pretendiente para obtener la mano de su elegida. Esta información suplementaria es útil para encontrar la serie de «declaraciones» cuando D y E se conviertan en pretendientes: cuando D se declara a 1, se encuentra la primera ruptura, porque 1 prefiere D a B. Entonces B se convierte en pretendiente y se declara a 2, lo que provoca la segunda ruptura porque 2 prefiere B a A. A se convierte en el nuevo pretendiente y hace su petición a 5, lo que proporciona una situación estable. ¡Pero esta estabilidad es sólo temporal! Como se puede comprobar en la serie de peticiones de mano que siguen a la llegada de E, las cosas no se estabilizan de nuevo hasta después de 8 peticiones. Mientras tanto se observa que E ha sido pretendiente dos veces.

El primer paso para una implementación consiste en imaginar una estructura de datos para las listas de preferencias. Éstas son sencillas listas enlazadas

como tipo de datos abstractos, pero se vio con los ejemplos del Capítulo 3 y en otros diferentes lugares que una buena elección de la representación puede tener un gran impacto sobre el rendimiento. Además, en el presente caso, se imponen estructuras diferentes para las listas de preferencia de hombres y las de las mujeres, porque las listas se utilizan de forma diferente.

Como los hombres recorren sus listas en secuencia, conviene perfectamente una implementación por lista lineal. Siendo las listas de preferencia todas del mismo tamaño, lo más sencillo es utilizar una representación por array de dos dimensiones. Por ejemplo, $\text{preferida}[h][m]$ será la m -ésima mujer de la lista de preferencia del h -ésimo hombre. Además se debe memorizar desde donde llega cada pretendiente en su lista. Esto se puede obtener por medio de un array de una dimensión siguiente , inicializado a 0 y tal que $\text{siguiente}[h]+1$ es el índice de la siguiente mujer de la lista de preferencia del hombre h ; el identificador de ella se encuentra en $\text{preferida}[h][\text{siguiente}[h]+1]$.

Se debe seguir la pista del novio de cada mujer ($\text{novio}[m]$ será el hombre elegido de la mujer m) y poder contestar a la pregunta: «¿Es el hombre p preferible al hombre $\text{novio}[m]$?» Se podría proceder recorriendo secuencialmente la lista de preferencia en cuestión hasta encontrar a p o a $\text{novio}[m]$, pero este método sería ineficaz si los dos se encuentran hacia el final de la lista. Hace falta la «inversa» de la lista de preferencia: $\text{rango}[m][p]$ es el índice del pretendiente p de la lista de preferencia de la mujer m . Para el ejemplo anterior se tiene que $\text{rango}[1][1]$ vale 2, por lo que A está en la segunda posición de la lista de preferencia de 1, $\text{rango}[5][4]$ vale 1 por lo que D está en la cuarta posición de la lista de preferencia de 5, etcétera.

Lo adecuado del pretendiente p se puede determinar rápidamente comprobando si $\text{rango}[m][p]$ es menor que $\text{rango}[m][\text{novio}[m]]$. Estos arrays se construyen fácilmente a partir de las listas de preferencia. Finalmente se utiliza un pretendiente inicial «centinela» 0 que se coloca al final de todas las listas de preferencia de las mujeres.

Con las estructuras de datos inicializadas de esta forma, la implementación antes descrita es inmediata:

```

for (h = 1; h <= N; h++)
{
    for (p = h; p != 0;)
    {
        siguiente[p]++;
        m = preferida[p][siguiente[p]];
        if (rango[m][p] < rango[m][novio[m]])
            { t = novio[m]; novio[m] = p; p = t; }
    }
}

```

Cada iteración comienza con un pretendiente sin novia y finaliza con una mujer sin novio. El bucle interno debe finalizar porque la lista de cada hombre

contiene a todas las mujeres, y cada iteración incrementa la lista de preferencia de algún hombre, y así se encuentra forzosamente una mujer sin novio antes de que todas las listas de preferencia de todos los hombres se hayan recorrido por completo. El conjunto de parejas formadas por el algoritmo es estable porque toda mujer a la que un hombre cualquiera podría preferir a su propia novia está ella misma emparejada con algún hombre al que ella prefiere.

Propiedad 34.2 *El problema de los matrimonios estables puede resolverse en tiempo lineal.*

Como se acaba de mencionar, cada iteración del bucle incrementa la lista de preferencia de alguno de los hombres. En el peor caso se examinan todas las entradas de las listas (pero nunca más de una vez). De hecho, el algoritmo puede llevar mucho menos tiempo que el que se necesita para construir las listas porque se puede encontrar una configuración estable bastante antes de haber terminado todas las listas. Este tipo de consideración conduce a un cierto número de problemas analíticos interesantes.■

Existen varios defectos muy relacionados con este algoritmo. En primer lugar los hombres eligen las mujeres de sus listas en orden, mientras que las mujeres deben esperar que se presente un «buen hombre». Se puede corregir este defecto (de forma más fácil que en la realidad) invirtiendo el orden en el que se entra en las listas de preferencia. Esta modificación da la configuración estable 1E 2D 3A 4C 5B en la que cada mujer consigue su primera elección, salvo 5 que obtiene la segunda. En general pueden existir muchas configuraciones estables: se puede mostrar que ésta es «óptima» para las mujeres en el sentido de que ninguna otra configuración estable dará a alguna mujer una elección mejor de su lista. (Por supuesto, la primera configuración estable del ejemplo es óptima para los hombres.)

Otra característica del algoritmo que parece ser un defecto es el orden en el que los hombres se convierten en pretendientes: ¿es mejor ser el primero en declararse (y así ser novio, aunque sea por poco tiempo, en primera elección) o ser el último (y así tener las mínimas posibilidades de sufrir la afrenta de una ruptura)? La respuesta es que no se trata del todo de un defecto: el orden en el que los hombres se convierten en pretendientes no tiene ninguna importancia. Mientras que cada hombre haga propuestas y que cada mujer acepte de acuerdo con sus listas de preferencia, se obtiene la misma configuración estable.

Algoritmos avanzados

Los dos casos que se acaban de examinar proporcionan alguna indicación sobre las dificultades del problema de la concordancia. Aunque estos algoritmos es-

pecíficos sean útiles en aplicaciones reales, como ya se ha dicho, muchas otras aplicaciones pueden necesitar la solución de problemas más generales.

Entre los problemas más generales que se han estudiado detalladamente están: el de la concordancia máxima para grafos cualesquiera (no sólo bipartidos); la concordancia ponderada para grafos bipartidos, en el que las aristas son los pesos y se busca una concordancia con una ponderación total máxima; y la concordancia ponderada para grafos en general.

La concordancia ponderada para grafos bipartidos y las generalizaciones similares se pueden tratar en la medida en que se conocen los algoritmos para la generalización de los problemas del flujo de red. Sin embargo, los grafos en general son otra historia. (El problema de los matrimonios estables puede caracterizarse como un medio para evitar el problema de la concordancia ponderada para grafos cualesquiera redefiniendo el problema.) El análisis de las numerosas técnicas que se han examinado llenaría un volumen completo: se trata de uno de los problemas más estudiados de la teoría de grafos.

Ejercicios

1. Encontrar todas las concordancias de cinco aristas del grafo bipartido de la Figura 34.2.
2. Utilizar el algoritmo dado en el texto para encontrar las concordancias máximas para grafos bipartidos de 50 vértices y 100 aristas. ¿Cuántas aristas contienen aproximadamente las concordancias?
3. Construir un grafo bipartido con seis nodos y ocho aristas que admita una concordancia de tres aristas o bien demostrar que una situación así es imposible.
4. Supóngase que los vértices de un grafo bipartido representan tareas y las aristas personas y que cada una de ellas debe realizar *dos* tareas. ¿La reducción de este problema al del flujo de red dará una solución de este tipo? Justificar la respuesta.
5. Modificar el programa del flujo de red del Capítulo 33 para que se aproveche de la estructura específica de redes 0-1 encontradas en la concordancia bipartida.
6. Escribir un algoritmo eficaz para determinar si un acuerdo es estable para el problema de los matrimonios.
7. ¿Es posible que dos hombres obtengan su última elección en el algoritmo del problema de matrimonios estables? Justificar la respuesta.
8. Construir un conjunto de listas de preferencia para $N = 4$ en el problema de los matrimonios estables, de tal forma que cada uno obtenga su segunda elección o demostrar que dicho conjunto no existe.
9. Presentar una configuración estable para el problema de los matrimonios estables en el caso en el que las listas de preferencia de los hombres y de las mujeres se presenten todas en orden creciente.

10. Ejecutar el problema de los matrimonios estables para $N = 50$ utilizando permutaciones aleatorias para las listas de preferencia. ¿Cuántas peticiones de mano se hacen aproximadamente durante la ejecución del algoritmo?

REFERENCIAS para Algoritmos sobre grafos

Existen varios manuales de algoritmos sobre grafos, pero el lector debe saber que hay mucho que aprender en lo que respecta a los grafos, que todavía no se ha llegado a comprenderlos en su totalidad y que tradicionalmente se han estudiado desde un punto de vista matemático (y no algorítmico). Por ello, muchas de las referencias proporcionan una descripción más rigurosa y profunda de muchos de los temas más difíciles que se han visto aquí.

Gran parte de los temas que se han presentado en esta sección se encuentran en los libros de Mehlhorn y Tarjan. Ambos son referencias básicas que proporcionan un tratamiento cuidadoso de los algoritmos sobre grafos, elementales y avanzados, así como referencias muy completas sobre los trabajos más actuales. Otra fuente de material complementario es el libro de Papadimitrou y Steiglitz. A pesar de que la mayor parte del libro trata de temas bastante más avanzados (por ejemplo, existe un tratamiento completo de la concordancia en grafos cualesquiera), cubre muchos de los algoritmos que se han presentado, incluyendo referencias a una gran cantidad de material.

La aplicación de la búsqueda en profundidad, para resolver problemas de conectividad y otros varios, es la obra de R. E. Tarjan, cuyo trabajo original merece un estudio a profundidad. Las numerosas variantes de los algoritmos para el problema de la unión-pertenencia del Capítulo 30 han sido clasificadas y comparadas por van Leeuwen y Tarjan. Los algoritmos para los caminos más cortos y árboles de expansión mínimos en grafos densos del Capítulo 31 son muy antiguos, pero los documentos originales de Dijkstra, Prim y Kruskal todavía constituyen una lectura interesante. El tratamiento del problema del matrimonio estable del Capítulo 34 está basado en la visión humorística proporcionada por Knuth.

- E. W. Dijkstra, «A note on two problems in connexion with graphs», *Numerische Mathematik*, **1** (1959).
- D. E. Knuth, *Marriages stables*, Les Presses de l'Université de Montréal, Montréal (1976).
- J. R. Kruskal Jr., «On the shortest spanning subtree of a graph and the traveling salesman problem», *Proceedings AMS*, **7**, 1 (1956).
- K. Mehlhorn, *Data Structures and Algorithms 2: NP-Completeness and Graph Algorithms*, Springer-Verlag, Berlín, 1984.
- C. H. Papadimitrou y K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ (1982).
- R. C. Prim, «Shortest connection networks and some generalizations», *Bell System Technical Journal*, **36** (1957).
- R. E. Tarjan, «Depth-first search and linear graph algorithms», *SIAM Journal on Computing*, **1**, 2 (1972).
- R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
- J. van Leeuwen y R. E. Tarjan, «Worst-case analysis of set-union algorithms», *Journal of the ACM*, 1986.

Algoritmos matemáticos

Números aleatorios

El próximo conjunto de algoritmos a tratar serán los métodos que se emplean para generar números aleatorios. Aunque ya se han encontrado tales números en diferentes contextos a lo largo del libro, se comenzará por intentar obtener una idea más exacta de lo que son.

Al hablar es frecuente utilizar el término *aleatorio* cuando realmente se quiere hacer referencia al término *arbitrario*. Cuando se pide un número *arbitrario*, se quiere decir que da lo mismo el número que se obtenga; casi cualquier número es válido. En cambio, un número *aleatorio* es un concepto matemático definido con precisión: cada número debe tener la misma probabilidad de ser generado. Un número aleatorio satisfará las necesidades de quien necesite un número arbitrario, pero no al contrario.

Para que la frase «cada número debe tener la misma probabilidad de ser generado» tenga sentido, se debe restringir el campo de los números a utilizar a un dominio finito. No se puede generar un número entero aleatorio cualquiera, únicamente un entero aleatorio dentro de un rango; no se puede generar un número real aleatorio, únicamente un número fraccionario aleatorio en un rango y con una precisión determinadas.

Casi nunca se necesita obtener un único número aleatorio, sino una *serie* de ellos (de lo contrario podría ser válido un número arbitrario). Aquí es donde intervienen las matemáticas: muchos hechos se pueden probar a partir de las propiedades de las series de números aleatorios. Por ejemplo, en una gran serie de números aleatorios en un dominio pequeño, se podría ver que cada valor aparece aproximadamente el mismo número de veces. Las series aleatorias modelan muchas situaciones naturales y se han descubierto muchas de sus propiedades. Para ser coherentes con los términos que se utilizan comúnmente, en este capítulo se designará a los números de las series aleatorias como números aleatorios.

No existe ningún modo de generar verdaderos números aleatorios en una computadora (o en cualquier sistema determinista). Una vez escrito el programa, los números que genera se pueden deducir, por lo que ¿cómo pueden

considerarse aleatorios? Lo mejor que se puede esperar es escribir programas que generen series de números que presenten la mayor parte de las propiedades de los números aleatorios. Dichos números se denominan normalmente *pseudo-aleatorios*: no son realmente aleatorios, pero se pueden considerar válidos como aproximación a los números aleatorios, de igual forma que los números con coma flotante son válidos como aproximación a los números reales. (Algunas veces es conveniente hacer una distinción adicional: en ciertos casos un pequeño número de propiedades de los números aleatorios son cruciales, mientras que otras son irrelevantes. En estos casos se puede generar números *cuasi-aleatorios* que con seguridad tienen las propiedades que interesan, siendo poco probable que tengan otras propiedades de los números aleatorios. En algunas aplicaciones, se puede demostrar que los números cuasi-aleatorios son preferibles a los números pseudo-aleatorios.)

Se puede comprobar fácilmente que la aproximación a la propiedad «cada número tiene la misma probabilidad de ser generado» no es suficiente en una gran serie. Por ejemplo, cada número del intervalo [1,100] aparece una vez en la serie (1, 2, ..., 100), pero esta secuencia tiene pocas posibilidades de ser válida como aproximación a una serie aleatoria. De hecho, en una serie aleatoria de 100 números de longitud en el intervalo [1,100] es probable que algunos números aparezcan más de una vez y otros no aparezcan nunca. Si no sucede esto en una serie de números pseudo-aleatorios entonces hay errores en el generador de números aleatorios. Se han desarrollado para los generadores de números aleatorios muchos métodos de comprobación sofisticados, basados en observaciones específicas con la finalidad de comprobar si una gran serie de números pseudo-aleatorios posee algunas de las propiedades de los números aleatorios. En este capítulo se verá con detalle uno de los más importantes, la prueba de χ^2 (*chi-cuadrado*).

Hasta ahora se ha hablado (y se continuará haciéndolo) exclusivamente de números aleatorios con distribución *uniforme*, en los que cada valor tiene la misma probabilidad. También es normal tratar con números aleatorios que obedezcan a otra distribución, en la que unos valores son más probables que otros. En general, los números pseudo-aleatorios con distribuciones no uniformes se obtienen llevando a cabo alguna operación sobre los uniformemente distribuidos. La mayoría de las aplicaciones de este libro emplean números aleatorios con distribución uniforme. Tal y como se verá, es bastante difícil convencerse de que los números generados tienen «todas» las propiedades de los números aleatorios; este problema adquiere especial relevancia cuando se tratan otros tipos de distribuciones.

Aplicaciones

En este libro se han encontrado muchas aplicaciones en las que se emplean números aleatorios, alguna de las cuales se recuerdan ahora. Una aplicación evi-

dente es la *criptografía*, cuyo principal objetivo es codificar un mensaje de forma que no pueda ser leído por nadie que no sea el destinatario. Como se vio en el Capítulo 23, una forma de hacerlo consiste en dar al mensaje una apariencia aleatoria, utilizando una serie pseudo-aleatoria para cifrarlo de tal forma que el receptor pueda utilizar la misma serie pseudo-aleatoria para descifrarlo.

Otro campo en el que se han empleado abundantemente los números aleatorios es la *simulación*. Una simulación típica incluye un gran programa que modela algún aspecto del mundo real: los números aleatorios son los datos normales para este tipo de programas. Incluso cuando no son necesarios verdaderos números aleatorios, las simulaciones suelen necesitar muchos números arbitrarios como datos de entrada que se obtienen oportunamente de un generador de números aleatorios.

Cuando se va a analizar una gran cantidad de datos a veces es suficiente con procesar un pequeño subconjunto de ellos, elegido de acuerdo con un *muestreo* aleatorio. Estas aplicaciones están muy difundidas, destacando entre ellas los sondeos de intención de voto en unas elecciones políticas.

Frecuentemente es necesario hacer una elección cuando todos los factores a considerar aparentan ser iguales. Los sorteos de la lotería nacional o los mecanismos utilizados en las residencias universitarias norteamericanas para decidir qué dormitorio corresponde a cada estudiante son ejemplos del empleo de los números aleatorios en la *toma de decisiones*. De esta forma, la responsabilidad de la elección se deja en manos del «destino» (o de la computadora).

Los lectores de este libro probablemente se encontrarán ellos mismos empleando números aleatorios en simulaciones, proporcionando a los programas datos de entrada aleatorios o arbitrarios. Se ha visto también que algunos algoritmos aumentan su eficacia al emplear números aleatorios para hacer muestras o para ayuda en la toma de decisiones. Los principales ejemplos de esto son la ordenación rápida (ver Capítulo 9) y el método de búsqueda de cadenas de Rabin-Karp (ver Capítulo 19).

Método de congruencia lineal

El método más conocido para generar números aleatorios, utilizado casi exclusivamente desde que fue introducido por D. Lehmer en 1951, es el denominado método de *congruencia lineal*. Si la variable *semilla* contiene algún número arbitrario, las siguientes instrucciones llenarán un array de *N* números aleatorios, utilizando este método:

```
a[0] = semilla;  
for (i = 1; i <= N; i++)  
    a[i] = (a[i-1]*b+1) % m;
```

Esto es, para obtener un nuevo número aleatorio, se toma el anterior, se multiplica por una constante b , se le suma 1 y se toma el resto de la división por una segunda constante m . El resultado es siempre un entero entre 0 y $m-1$. Esta técnica es de interés para su empleo en computadoras, porque la implementación de la función `%` suele ser trivial; si se ignora el desbordamiento de capacidad en las operaciones aritméticas, ya que la mayoría del hardware de las computadoras desecha los bits desbordados, efectivamente se desarrolla una operación `%` con m igual a una unidad mayor que el entero más grande que se pueda representar por una palabra de la computadora. Una vez más, los números no son realmente aleatorios; el programa tan sólo genera números de los que se espera que *parezcan* aleatorios en algún otro proceso.

A pesar de lo simple que pueda parecer, el generador de números aleatorios por congruencia lineal ha sido objeto de volúmenes enteros de estudio detallado y difíciles análisis matemáticos. Este trabajo proporciona algunas guías para elegir las constantes `semilla`, `b` y `m`. Algunas de las principales aplicaciones son de «sentido común», pero en este caso el sentido común no es suficiente para garantizar buenos números aleatorios. Primero, m tiene que ser grande: puede ser del tamaño de una palabra de la computadora, como se dijo anteriormente, pero si no conviene no es necesario que sea tan grande (ver la implementación posterior). Por lo regular, será conveniente hacer que m sea una potencia de 10 o de 2. Segundo, `b` no debe ser ni muy grande ni muy pequeño: una elección razonable es emplear un número con un dígito menos que m . Tercero, `b` debe ser una constante arbitraria sin ningún tipo de estructura en sus dígitos, *excepto* que debe terminar en ...x21, con x par: hay que reconocer que este último requisito es peculiar pero permite evitar la aparición de algunos casos problemáticos que no han sido descubiertos por el análisis matemático.

Las reglas antes descritas fueron desarrolladas por D.E. Knuth, cuyo libro trata el tema en detalle. Knuth demuestra que este método de congruencia lineal produce buenos números aleatorios que superan muchas pruebas estadísticas sofisticadas. El problema potencial más serio, que puede aparecer de forma rápida, es que el generador puede verse atrapado en un ciclo y generar, antes de lo que debiera, números ya generados antes. Por ejemplo, la elección $b=19$, $m=381$ y `semilla=0`, produce la serie 0, 1, 20, 0, 1, 20, ..., una secuencia de enteros entre 0 y 380 no muy aleatoria. Desgraciadamente no todas las dificultades son tan fáciles de detectar, por lo que es un buen consejo seguir las directrices recomendadas por Knuth, evitando así las sutiles trampas que él descubrió.

Se puede utilizar cualquier valor inicial para comenzar la generación de los números aleatorios sin ninguna consecuencia (excepto, por supuesto, qué valores iniciales diferentes producen sucesiones aleatorias distintas). Con frecuencia es necesario almacenar la serie completa, como en el programa anterior. En lugar de ello se puede utilizar una variable global `a`, inicializada con algún valor, y actualizada posteriormente por medio de la operación $(a*b + 1) \% m$.

En C++ (y en muchos otros lenguajes de programación) todavía falta un paso antes de llegar a una implementación operativa porque no se puede ignorar el desbordamiento, que es una condición de error que puede conducir a resultados

imprevisibles. Se supone que la computadora tiene una palabra de 32 bits y que se elige $m=100000000$, $b=31415821$, e, inicialmente, $a=1234567$. Todos estos valores son claramente inferiores al entero más grande que se puede representar, pero la primera operación $a*b+1$ produce un desbordamiento. La parte del producto que causa el desbordamiento no es relevante para este proceso; solamente interesan los últimos ocho dígitos. El truco está en evitar el desbordamiento mediante la división de la multiplicación en dos partes. Para multiplicar p por q , se escribe $p = 10^4 p_1 + p_0$ y $q = 10^4 q_1 + q_0$ por lo que el producto es:

$$\begin{aligned} pq &= (10^4 p_1 + p_0)(10^4 q_1 + q_0) \\ &= 10^8 p_1 q_1 + 10^4(p_1 q_0 + p_0 q_1) + p_0 q_0. \end{aligned}$$

Ahora solamente se quieren ocho dígitos del resultado, por lo que se puede ignorar el primer término y los cuatro primeros dígitos del segundo. Esto conduce al siguiente programa:

```
#include <iostream.h>
const int m = 100000000;
const int m1 = 10000;
int mult(int p, int q)
{
    int p1, p0, q1, q0;
    p1 = p/m1; p0 = p%m1;
    q1 = q/m1; q0 = q%m1;
    return (((p0*q1+p1*q0) % m1)*m1+p0*q0) %m;
}
class Aleatorio
{
private:
    int a;
public:
    Aleatorio(int semilla)
    { a = semilla; }
    int siguiente ()
    {const int b = 31415821;
     a = (mult(a, b) + 1) % m;
     return a;
    }
};
main()
{
    int i, N; Aleatorio x(1234567);
```

```

    cin >> N;
    for (i = 1; i<= N; i++)
        cout << x.siguiente() << ' ';
    cout << '\n';
}

```

La función `mult` de este programa calcula $p*q \% m$, sin desbordamiento mientras que m sea inferior a la mitad del mayor entero que se pueda representar. La técnica se puede aplicar, evidentemente, con $m=m_1*m_1$ para otros valores de m_1 .

Cuando se ejecuta el programa con $N = 10$ y $a = 1234567$, genera los diez números siguientes: 35884508, 80001069, 63512650, 43635651, 1034472, 87181513, 6917174, 209855, 67115956, 59939877. Desde luego, esta serie no es aleatoria: por ejemplo, los últimos dígitos toman valores del 0 al 9 cíclicamente. Es fácil comprobar que sucederá esto a partir de la fórmula. En general, los dígitos de la derecha no son aleatorios, un hecho que es el origen de un error importante y bastante común en este tipo de generadores de números aleatorios. Por lo regular, en las aplicaciones se desean números aleatorios en el intervalo $[0, r]$, donde r es un entero que depende de una aplicación y no puede ser convenientemente integrado en el generador de números aleatorios, como M en el programa anterior.

El siguiente es un sencillo ejemplo de lo que es un mal programa para generar números aleatorios en un intervalo pequeño:

```

int malsiguiente(int r)
{ const int b = 31415821;
  a = (mult(a, b) + 1) % m;
  return a % r;
}

```

Los únicos dígitos que se utilizan son los no aleatorios de la derecha por lo que la serie resultante posee pocas de las propiedades deseadas. Este problema se resuelve fácilmente utilizando en su lugar los dígitos incluso de la *izquierda*. Se desea calcular un número entre 0 y $r-1$ por medio de $(a*r)/m$, pero de nuevo hay que evitar el desbordamiento, como en la siguiente implementación:

```

int siguiente(int r)
{ const int b = 31415821;
  a = (mult(a, b) + 1) % m;
  return ((a/m1)*r)/m1;
}

```

Con este programa se puede tomar $r = 2$ para obtener un flujo binario, o $r = 10$ para obtener dígitos decimales, o $r = N$ para obtener un elemento de partición de una clasificación rápida, etcétera.

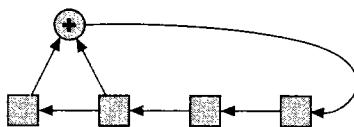


Figura 35.1 Registro de retroalimentación lineal de cuatro bits.

Otra técnica que suele utilizarse consiste en generar números aleatorios reales entre 0 y 1 considerando los números anteriores como fracciones con la coma a la izquierda (precedida de 0). Eso se puede implementar fácilmente devolviendo el valor real a/m en lugar del entero a . Así un usuario puede obtener un entero del intervalo $[0, r)$ simplemente multiplicando este valor por r y truncando al entero más próximo. O lo que se necesite exactamente quizás sea un número real aleatorio entre 0 y 1.

Método de congruencia aditiva

Otro método para generar números aleatorios está basado en los registros particulares de retroalimentación lineal, método utilizado en las primeras máquinas de codificar. La idea es comenzar por un registro que contenga un patrón arbitrario (que no sean todos ceros) y modificarlo avanzando en cada paso una posición hacia la derecha, llenando los espacios que quedan vacíos a la izquierda con un bit determinado por el contenido del registro.

La Figura 35.1 muestra un simple registro de retroalimentación lineal de cuatro bits, en el que el nuevo bit se ha obtenido tomando el «o exclusivo» de los dos bits más a la derecha. Por ejemplo, si el registro contiene inicialmente el patrón 1111, entonces contendrá 0111 después del primer paso: los bits 111 se mueven a la derecha una posición y el bit más a la izquierda se convierte en 0 porque los dos más a la derecha eran iguales. Continuando así, el registro contiene sucesivamente 0011, 0001, 1000, 0100, 0010, 1001, 1100, 0110, 1011, 0101, 1010, 1101, 1110, 1111. Una vez que se encuentra el modelo inicial, el proceso se repite.

Hay que tener en cuenta que aparecerán todos los posibles modelos que tienen algún bit distinto de cero: el valor inicial se repite cada 15 pasos. De todas formas, si se cambian las posiciones «clave» (los bits utilizados para la retroalimentación) de 0 a 2 (comenzando por la derecha) en vez de 0 a 1, se obtiene la serie: 1111, 0111, 0011, 1001, 1100, 1110, 1111, no el ciclo completo. Sería deseable garantizar que siempre se obtenga un ciclo completo.

En general, para un registro de n bits modificado por retroalimentación lineal, es posible organizarlo para que el ciclo tenga una longitud de $2^n - 1$. Ade-

más, para un gran valor de n los registros proporcionan buenos generadores de números aleatorios. Por lo regular se puede utilizar $n = 31$ o $n = 63$. Como en el método de congruencia lineal, las propiedades matemáticas de estos registros se han estudiado ampliamente. Por ejemplo, se han descubierto bastantes relaciones entre la selección de las diferentes posiciones «clave» y la generación de todos los patrones de bits para registros de distintos tamaños. Por ejemplo, para $n = 31$, serían válidas las posiciones clave 0, e incluso 4, 7, 8, 14, 19, 25, 26 o 29.

El contenido del registro no es útil como serie aleatoria, porque todos los bits menos uno se superponen en cada par sucesivo. Es preferible considerar este dispositivo como un generador de una secuencia de bits aleatorios (los bits del registro más próximo a la izquierda), en el ejemplo 1000100110110111. Como se mencionó en el Capítulo 23, tales dispositivos se emplean en criptografía porque pueden generar largas series de bits a partir de claves pequeñas.

Otro hecho interesante es que se puede calcular una palabra de la computadora de una sola vez, en lugar de bit a bit, empleando la misma fórmula de recursión. En el ejemplo, si se toma a modo de bit el «o exclusivo» de dos palabras sucesivas, se obtiene la palabra que aparece tres lugares más adelante en la lista. Esto conduce a desarrollar un generador de números aleatorios de fácil implementación en una computadora de uso generalizado. La utilización de un registro de retroalimentación con los bits en las posiciones clave b y c corresponde al empleo de la recursión: $a[k] = (a[k-b] + a[k-c]) \% m$. Para guardar la correspondencia con el modelo de registro desplazado, el «+» de la recursión debería reemplazarse por un «o exclusivo». De todas formas, se ha demostrado que los buenos números aleatorios se pueden generar incluso empleando una suma normal de enteros. Éste es el método denominado *congruencia aditiva*.

El bit más próximo a la derecha de los números producidos por un generador de congruencia aditiva se comporta exactamente igual que el bit correspondiente del cambio de registros por retroalimentación lineal, por lo que el número de pasos necesarios para que el método comience a repetirse es al menos tan grande como la longitud del ciclo. Salvo este hecho, se han desarrollado pocos resultados específicos sobre los números producidos por dichos generadores: la evidencia en su favor es eminentemente empírica (superan sin problemas las pruebas estadísticas).

Para implantar un generador de congruencia aditiva se necesita mantener una tabla de tamaño c que contenga siempre los últimos c números generados. El cálculo comienza por reemplazar uno de los números de la tabla por la suma de otros dos, como en la siguiente implementación:

```
class Aleatorio
{
    private:
        ini a[55], j;
public:
```

```

Aleatorio(int semilla);
int siguiente(int r)
{
    j = (j+1) % 55;
    a[j] = (a[(j+23) % 55]+a[(j+54) % 55]) % m;
    return ((a[j]/m1)*r)/m1;
}
};

```

La variable global *a* se ha reemplazado por una tabla completa con un puntero (*j*) en ella. Knuth recomienda elegir *b* = 31, *c* = 55 para la recurrencia en el método de congruencia aditiva, y así se memorizan los 55 números generados más recientemente. La estructura de datos apropiada para esto es una cola (ver Capítulo 3), pero como debe ser de tamaño fijo, se utiliza un array de ese tamaño indexado por un puntero «circular». Esta gran cantidad de «estados globales» es un inconveniente de este generador en algunas aplicaciones, pero también es una ventaja porque conduce a un ciclo extremadamente largo (al menos $2^{55}-1$, incluso si el módulo *m* no es grande).

La tabla debe inicializarse con números que no sean demasiado pequeños ni demasiado grandes. Una forma fácil de conseguir estos números es utilizar un simple generador de congruencia lineal:

```

Aleatorio:: Aleatorio(int semilla)
{ const int b = 31415821;
  a[0] = semilla;
  for (j = 1; j <= 54; j++)
    a[j] = (mult(a[j-1], b) + 1) % m;
}

```

Este programa devuelve un entero aleatorio entre 0 y *r*-1. Se puede modificar fácilmente, como ya se mencionó, para obtener una función que devuelva un número aleatorio real entre 0 y 1 (*a*[*j*] /*m*).

Comprobación de la aleatoriedad

Con frecuencia se puede detectar que una serie no es aleatoria, pero certificar que *es* aleatoria es verdaderamente difícil. Como se mencionó con anterioridad, ninguna serie generada por una computadora puede ser de hecho aleatoria, pero se puede conseguir una que presente muchas de las propiedades de los números aleatorios. Por desgracia, a menudo es imposible precisar qué propiedades de los números aleatorios son importantes para una aplicación en particular. Ade-

más siempre es una buena idea aplicar algún tipo de comprobación al generador de números aleatorios para asegurarse de que no han aparecido soluciones degeneradas. Los generadores de números aleatorios pueden ser muy, muy buenos, pero cuando son malos, son pésimos.

Se han desarrollado muchos métodos de comprobación para determinar si una serie comparte ciertas propiedades con una verdadera serie aleatoria. La mayoría de ellos tienen una sólida base matemática y examinarlos detalladamente está fuera del alcance de este libro. Sin embargo, una prueba estadística, la prueba de χ^2 (chi-cuadrado) es de naturaleza fundamental, bastante fácil de implementar y útil en diversas aplicaciones, por lo que se examinará con alguna precisión.

La idea de la prueba de χ^2 es verificar si los números generados se distribuyen de forma razonablemente extendida. Si se generan N números positivos menores que r se podría esperar obtener aproximadamente N/r números de cada clase. Pero, y ésta es la esencia del método, las frecuencias de aparición de todos los valores no deben ser exactamente iguales: ¡eso no sería aleatorio! Así que no es difícil calcular si una serie de números está distribuida como una serie aleatoria: calcular la suma de los cuadrados de las frecuencias de aparición de cada valor, dividida por la frecuencia esperada, y después restar el tamaño de la serie. Este número, el «estadístico χ^2 », se puede expresar matemáticamente como

$$\chi^2 = \frac{\sum_{0 \leq i < r} (f_i - N/r)^2}{N/r},$$

y es fácilmente calculable, como en el siguiente programa:

```
float chicuadrado (int N, int r)
{
    int i, t, f[rmax]; Aleatorio x(1234567);
    for (i = 0; i < r; i++) f[i] = 0;
    for (i = 0; i < N; i++) f[x.siguiente(r)]++;
    for (i = 0, t = 0; i < r; i++) t += f[i]*f[i];
    return (float) ((r*t/N) - N);
}
```

Si el χ^2 estadístico está próximo a r , los números son aleatorios; si está demasiado alejado, no lo son. Las nociones de «próximo» y «alejado» se pueden definir con mayor precisión: hay tablas que indican exactamente cómo relacionar el estadístico con las propiedades de las series aleatorias. Para la sencilla prueba que se está aplicando, el estadístico debe estar a menos de $2\sqrt{r}$ de r . Esto es válido si N es mayor que aproximadamente $10r$; para estar seguros, se debe repetir la prueba varias veces, porque puede fallar una de cada diez.

Esta prueba es tan fácil de implementar que es razonable incluirla en todos

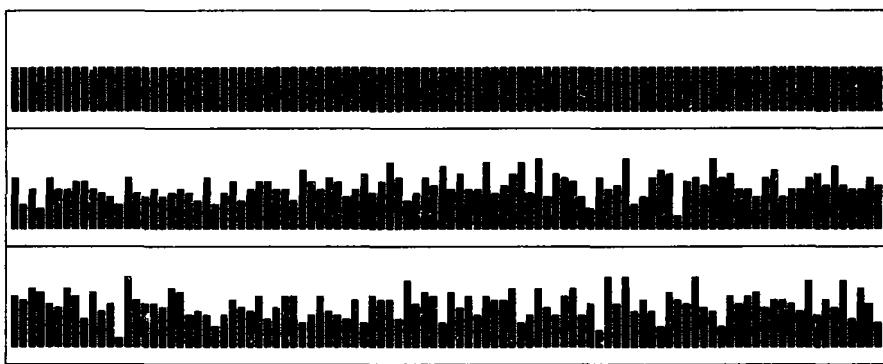


Figura 35.2 Frecuencias de tres generadores: bits de derecha, bits de izquierda y mal multiplicador.

los generadores de números aleatorios, aunque sólo sea para comprobar que nada inesperado puede causar problemas serios. Todos los «buenos» generadores que se han analizado superan esta prueba; los «malos» no. Si se utilizan los generadores anteriores para generar mil números menores que 100, se obtiene un estadístico χ^2 de 100,8 para el método de congruencia lineal y de 105,4 para el método de congruencia aditiva, ambos dentro de 100 ± 20 . Pero para el «mal» generador que utiliza el bit de la parte derecha del generador de congruencia lineal, se obtiene un valor de 0 (¿por qué?) y para el método de congruencia lineal con un mal multiplicador (101011) se obtiene un valor de 77,8, que está significativamente fuera del margen. La Figura 35.2 muestra la frecuencia de aparición de cada valor menor que 100 para las tres versiones del método de congruencia lineal que se acaban de mencionar. Aunque la utilización de los bits del lado izquierdo (diagrama superior) es obviamente desaconsejable, no es fácil distinguir la diferencia entre los diagramas de frecuencias inferior y central: el estadístico χ^2 facilita el camino para identificar el mal multiplicador.

Notas de implementación

Por lo regular se añade un conjunto de opciones para hacer que los generadores de números aleatorios sean más útiles en ciertas aplicaciones. A menudo es preferible construir el generador en forma de función, a la que se inicializa y después se llama repetidamente, generando cada vez un número aleatorio diferente. Otra posibilidad es llamar una sola vez al generador de números aleatorios y hacer que llene un array con todos los números aleatorios necesarios para un cálculo dado. En ambos casos es deseable que el generador produzca la misma serie en sucesivas llamadas (para la depuración inicial o en comparaciones de

programas con las mismas entradas) y que produzca una serie arbitraria (para la última depuración). Estas opciones suponen la modificación del «estado» memorizado entre cada dos generaciones de números aleatorios. Esto puede ser incómodo en algunos entornos de programación, pero el anterior programa en C++ puede adaptarse fácilmente a la implementación de dichas facilidades como funciones de clase adicionales. El generador aditivo presenta la desventaja de tener un estado relativamente largo (el array de palabras recientemente generadas), pero tiene la ventaja de disponer de un ciclo tan largo que es verdaderamente innecesario que los usuarios deban inicializarlo.

Una forma conservadora de protegerse de las excentricidades eventuales de un generador de números aleatorios consiste en combinar dos generadores diferentes. (La utilización del generador de congruencia lineal para inicializar la tabla relativa al método de congruencia aditiva es un ejemplo elemental de esta técnica.) Una forma fácil de implementar dos generadores combinados es emplear el primero para llenar una tabla y con el segundo elegir posiciones aleatorias de la tabla para extraer valores de salida (y almacenar los nuevos números proporcionados por el primer generador).

Cuando se depura un programa que emplea un generador de números aleatorios normalmente es una buena idea utilizar al principio un generador degenerado o trivial, tal como uno que devuelve siempre el valor 0, o uno que devuelve una serie de números en orden.

Por regla general, los generadores de números aleatorios son delicados y necesitan ser tratados con respeto. Es difícil estar seguro de que un generador en concreto es bueno mientras no se le someta a una enorme cantidad de diversas comprobaciones estadísticas. Lo honrado es hacer lo posible para utilizar un buen generador basado en el análisis matemático y en la experiencia de los demás; para estar seguro, hay que examinar los números para asegurarse de que «parezcan» aleatorios; si alguna cosa va mal, ¡cúlpese al generador!

Ejercicios

1. Escribir un programa para generar palabras aleatorias de cuatro letras (colecciones de letras). Estímese cuántas palabras generará el programa antes de repetir una palabra.
2. ¿Cómo se podría simular, generando números aleatorios, el lanzamiento de dos dados y haciendo luego la suma de los números obtenidos, sabiendo que los dados no son estándar (por ejemplo, sus caras tienen los números 1, 2, 3, 5, 8 y 13)?
3. Mostrar una secuencia de patrones producidos por un cambio de registros mediante una retroalimentación lineal como en la Figura 35.1, pero con las posiciones clave en el primero y último bits. Se supone que el patrón inicial es 1111.

4. ¿Por qué los operadores «o» e «y» no son válidos (como la función «o exclusivo») para el cambio de registro por retroalimentación lineal?
5. Escribir un programa para generar una imagen aleatoria bidimensional. (Por ejemplo: generar bits aleatorios, escribir un «*» cuando se genere un 1, « » cuando se genere un 0. Otro ejemplo: emplear números aleatorios como coordenadas de un sistema cartesiano bidimensional y escribir un «*» en los puntos designados.)
6. Emplear un generador de congruencia aditiva de números aleatorios para generar 1.000 enteros positivos menores que 1.000. Diseñar y aplicar una prueba para determinar si son o no aleatorios.
7. Emplear un generador de congruencia lineal con parámetros a elegir para generar 1.000 enteros positivos menores que 1.000. Diseñar y aplicar una prueba para determinar si son o no aleatorios.
8. ¿Por qué sería imprudente emplear, por ejemplo, $b = 3$ y $c = 6$ en el generador de congruencia aditiva?
9. ¿Qué valor genera el estadístico χ^2 para un generador degenerado que devuelve siempre el mismo número?
10. Describir cómo se pueden generar números aleatorios con m mayor que el tamaño de una palabra de la computadora.

Aritmética

A pesar de que la situación está en proceso de cambio, la *razón de ser* de muchos sistemas de computadoras es su capacidad para hacer rápidamente cálculos numéricos correctos. Las computadoras tienen capacidad por sí mismas para ejecutar operaciones sobre enteros y números reales representados con coma flotante; por ejemplo, C++ permite números del tipo `integer` o `float`, con todas las operaciones aritméticas normales definidas en ambos tipos. El método utilizado realmente para las operaciones aritméticas es parte de la arquitectura de la computadora y no interesa aquí (aunque un nuevo algoritmo rápido para, por ejemplo, la multiplicación de dos enteros de 32 bits realmente podría ser de gran importancia). En su lugar, se tratarán algunos de los algoritmos que entran en juego cuando las operaciones se deben ejecutar sobre objetos matemáticos más complicados.

En este capítulo se considerará la implementación en C++ de algoritmos para la suma y multiplicación de polinomios, de enteros muy grandes y de matrices. Los algoritmos elementales para estos problemas son comunes y simples, pero merece la pena reflexionar sobre cómo adaptar las estructuras de datos básicas a situaciones particulares. Esta vez el énfasis en las aplicaciones será menor de lo usual; en su lugar se considerará la complejidad de cálculo de problemas aritméticos fundamentales. Por ejemplo, a pesar de ser natural la implementación de algoritmos en el contexto de las clases de C++ para polinomios, grandes enteros y matrices, se evita hacerlo así con el fin de concentrar el interés en las características esenciales de los algoritmos. Por otra parte, las aplicaciones prácticas que puedan beneficiarse de los métodos aquí considerados son algo menos numerosas que las de algunos otros problemas que se han estudiado y, además, el estudio de algoritmos para llevar a cabo operaciones aritméticas básicas en objetos fundamentalmente matemáticos raras veces necesita una mayor justificación.

La aritmética proporciona un excelente ejemplo de cómo una correcta aplicación de los algoritmos puede producir métodos sofisticados que son sustancialmente más eficaces (asintóticamente) que los métodos elementales. Tal in-

vestigación es interesante, no sólo por lo que dice sobre la naturaleza de estos problemas esencialmente operativos, sino por su contexto histórico: los algoritmos para realizar operaciones aritméticas elementales tales como la suma, multiplicación y división son muy antiguos, remontándose a los orígenes de los estudios sobre algoritmos en el trabajo del matemático árabe al-Khowarizmi, cuyas raíces son más antiguas, del tiempo de los antiguos griegos y babilonios.

Se verá que la multiplicación polinómica y la multiplicación de matrices proporcionan ejemplos clásicos de la fuerza del paradigma de divide y vencerás. Desgraciadamente, los algoritmos resultantes (con la significativa excepción del método presentado en el Capítulo 41) son de difícil aplicación; los métodos elementales que utilizan estructuras básicas de datos son mejores, excepto para problemas cuya longitud sea excesiva.

Aritmética polinómica

Supóngase que se desea escribir un programa para sumar dos polinomios; debería realizar cálculos cómo

$$(1 + 2x - 3x^3) + (2 - x) = 3 + x - 3x^3.$$

En general, supóngase que se desea que el programa sea capaz de calcular $r(x) = p(x) + q(x)$, donde p y q son polinomios con N coeficientes. Esto se obtiene fácilmente con una representación por array. Se representa al polinomio $p(x) = p_0 + p_1x + \dots + p_{N-1}x^{N-1}$ por el array $p[N]$ con $p[j] \equiv p_j$, etc. Entonces la suma no es más que el programa de una sola línea

```
for (i = 0; i < N; i++) r[i] = p[i]+q[i];
```

Como es habitual en C++ (Capítulo 3), hay que conocer con anterioridad cuál es el mayor valor que podría tomar N , dado que los tamaños de los arrays p , q y r deben ser iguales al mayor de los tamaños previstos.

El programa anterior muestra que la suma es bastante fácil una vez que se ha hecho la representación de los polinomios por array; las otras operaciones son también fáciles de codificar. Por ejemplo, el siguiente fragmento de código es una implementación inmediata de la multiplicación de polinomios:

```
for (i = 0; i < 2*N-1; i++) r[i] = 0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        r[i+j] += p[i]*q[j];
```

La declaración de r debe permitir dos veces más coeficientes para el producto.

Cada uno de los N coeficientes de p se multiplica por cada uno de los N coeficientes de q , así que el tiempo de ejecución de este algoritmo es claramente cuadrático en el número de coeficientes.

Como se vio en el Capítulo 3, una de las ventajas de la representación de polinomios por un array que contiene sus coeficientes es que resulta más fácil hacer referencia a cualquier coeficiente directamente; una desventaja es que hay que reservar memoria para más números de los necesarios. Por ejemplo, el programa anterior no se podría utilizar razonablemente para multiplicar

$$(1 + x^{10000})(1 + 2x^{10000}) = 1 + 3x^{10000} + 2x^{20000},$$

aunque los datos impliquen solamente cuatro coeficientes y el resultado tres.

Otra solución sería representar los polinomios utilizando listas enlazadas y sumando como sigue:

```
struct nodo *sumar(struct nodo *p, struct nodo *q)
{
    struct nodo *t;
    t = z; z->c = 0;
    while ((p != z) && (q != z))
    {
        t->siguiente = new nodo;
        t = t->siguiente; t->c = p->c + q->c;
        p = p->siguiente; q = q->siguiente;
    }
    t->siguiente = z; t = z->siguiente; z->siguiente =z;
    return t;
}
```

Los polinomios de la entrada están representados por listas enlazadas a razón de un elemento de la lista por coeficiente; los polinomios de la salida están construidos por el procedimiento *sumar*. Las manipulaciones de los enlaces son muy similares a los programas que se vieron en los capítulos 3, 8, 14, 29 y en otras partes de este libro.

De esta forma, el programa anterior no constituye una mejora real con respecto a la representación por array, excepto que permite evitar los problemas de gestión de arrays dinámicos en C++ (con el coste de espacio de utilizar un enlace por coeficiente). Sin embargo, como sugería el ejemplo anterior, es posible aprovecharse de la posibilidad de que algunos de los coeficientes puedan ser cero. Se puede hacer que los nodos de la lista representen sólo los términos no nulos del polinomio, incluyendo también el grado de cada término representado por los nodos de lista, de forma que cada uno de ellos contenga los valores de c y j para representar cx^j . Entonces es conveniente separar la función de

creación e inserción de un nodo en una lista, como en la implementación siguiente:

```
struct nodo
    { int c; int j; struct nodo *siguiente; };
struct nodo *insertar(struct nodo *t, int c, int j)
{
    t->siguiente = new nodo;
    t = t->siguiente; t->c = c; t->j = j;
    return t;
}
```

La función `insertar` crea un nuevo nodo, le atribuye campos específicos y lo enlaza en una lista después del nodo `t`. Para hacer posible el procesamiento de polinomios de forma organizada, los nodos de lista pueden almacenarse en orden creciente del grado del término representado.

Ahora la función `sumar` comienza a ser más interesante, dado que se ha llevado a cabo una suma sólo de los términos del mismo grado y se asegura que no aparece ningún término de coeficiente cero en la salida:

```
struct nodo *sumar(struct nodo *p, struct nodo *q)
{
    struct nodo *t;
    t = z; z->c = 0; z->j = maxN;
    while ((p !=z) || (q !=z))
    {
        if ((p->j == q->j) && ((p->c + q->c) != 0))
        {
            t = insertar(t, p->c+q->c, p->j);
            p = p->siguiente; q = q->siguiente;
        }
        else if (p->j < q->j)
            { t = insertar(t, p->c, p->j); p = p->siguiente; }
        else if (q->j < p->j)
            { t = insertar(t, q->c, q->j); q = q->siguiente; }
    }
    t->siguiente = z; t = z->siguiente; z->siguiente = z;
    return t;
}
```

Estas mejoras son útiles para tratar polinomios «dispersos» con muchos coeficientes nulos, porque esto significa que la memoria y el tiempo necesarios para

procesar los polinomios serán proporcionales al número de coeficientes, no al grado del polinomio. Se pueden conseguir economías similares para otras operaciones sobre polinomios, por ejemplo la multiplicación, pero hay que tener precaución porque los polinomios pueden hacerse cada vez menos dispersos después de efectuar un cierto número de tales operaciones. La representación por array es la mejor si solamente existen unos pocos términos con coeficientes cero, o si el grado no es elevado. Para simplificar, se ha supuesto que ésta será la representación de los polinomios descritos posteriormente.

Un polinomio puede tener no sólo una, sino varias variables. Por ejemplo, se podrían tener que procesar polinomios tales como

$$1 + wx^2 + y^6z + w^{25}x^{50}y^{99}z^{38} + x^{1000}z^{1000}.$$

En estos casos la representación por lista enlazada es definitivamente imprescindible; la alternativa (arrays multidimensionales) necesitaría demasiado espacio. No es difícil generalizar el programa anterior sumar (por ejemplo) para manejar tales polinomios.

Evaluación e interpolación polinómica

A continuación se va a tratar de calcular el valor de un polinomio dado en un punto también dado. Por ejemplo, para evaluar

$$p(x) = x^4 + 3x^3 - 6x^2 + 2x + 1$$

para cualquier x dada, se podría calcular x^4 , después calcular y añadir $3x^3$, etc. Este método necesita calcular las potencias de x ; alternativamente, se podrían almacenar las potencias de x a medida que se van calculando, pero esto requiere memoria extra.

Un método simple que evita el cálculo y no utiliza espacio extra es el conocido como la *regla de Horner*: alternando apropiadamente las operaciones de suma y multiplicación, se puede evaluar un polinomio de grado N utilizando solo $N-1$ multiplicaciones y N sumas. La escritura entre paréntesis

$$p(x) = x(x(x(x + 3) - 6) + 2) + 1$$

hace evidente el orden de cálculo:

```
y = p[N];
for (i = N-1; i >= 0; i--) y = x*y + p[i];
```

Se ha utilizado ya una versión de este método en una aplicación práctica muy

importante, el cálculo de funciones de dispersión de grandes claves (ver Capítulo 16).

Un problema más complicado es evaluar un polinomio dado en diferentes puntos, para el que son apropiados diversos algoritmos, dependiendo de cuántas evaluaciones se van a hacer y si se van a efectuar simultáneamente o no. Si se debe hacer un gran número de evaluaciones, puede merecer la pena efectuar algunos «precálculos» que pueden reducir un tanto el coste de evaluaciones posteriores. Hay que destacar que el método de Horner necesita unas N^2 multiplicaciones para evaluar un polinomio de grado N en diferentes puntos. Se han diseñado métodos mucho más sofisticados para poder resolver el problema en $N(\log N)^2$ pasos, y en el Capítulo 41 se verá uno que utiliza solamente $N \log N$ multiplicaciones para un conjunto específico de N puntos de interés.

Si el polinomio dado tiene sólo un término, el problema de evaluación del polinomio se reduce a una *exponenciación*: calcular x^N . La regla de Horner degenera en este caso a un algoritmo trivial que necesita $N - 1$ multiplicaciones. Para ver que es posible hacerlo mucho mejor, se considera la siguiente serie para calcular x^{32} :

$$x, x^2, x^4, x^8, x^{16}, x^{32}.$$

Cada término se obtiene elevando al cuadrado el término anterior, así que sólo se necesitan cinco multiplicaciones, no treinta y uno.

El método de los «cuadrados sucesivos» se puede generalizar fácilmente a un N cualquiera si se memorizan los valores calculados. Por ejemplo, x^{55} se puede calcular a partir de la evaluación anterior con cuatro multiplicaciones más:

$$x^{55} = x^{32}x^{16}x^4x^2x^1$$

En general, se puede utilizar la representación binaria de N para elegir qué valores calculados utilizar. (En el ejemplo, como $55 = (110111)_2$, se utilizan todos menos x^8 .) Se pueden calcular los cuadrados sucesivos y examinar los bits de N con el mismo bucle. Existen dos métodos para implementar esto utilizando un solo «acumulador», como en el método de Horner. Uno de los algoritmos implica examinar la representación binaria de N de izquierda a derecha, empezando con un 1 en el acumulador. En cada paso se eleva al cuadrado el acumulador y también se multiplica por x cuando hay un 1 en la representación binaria de N . La siguiente serie de valores está calculada por este método para $N = 55$:

$$1, 1, x, x^2, x^3, x^6, x^{12}, x^{26}, x^{27}, x^{54}, x^{55}.$$

Otro algoritmo muy conocido funciona de forma similar, pero examina N de derecha a izquierda. Este problema es un ejercicio estándar de introducción a la programación. Aunque parece que tiene poco interés práctico para calcular números tan grandes, se verá posteriormente en la presentación de grandes enteros

que este método desempeña un papel fundamental en la implementación de sistemas de cripto de claves públicas del Capítulo 23.

El problema «inverso» al de evaluación simultánea de un polinomio de grado N en N puntos es el de la *interpolación polinómica*: dado un conjunto de N puntos x_1, x_2, \dots, x_N y los valores asociados y_1, y_2, \dots, y_N , encontrar el único polinomio de grado $N - 1$ tal que

$$p(x_1) = y_1, p(x_2) = y_2, \dots, p(x_N) = y_N.$$

El problema de la interpolación consiste en encontrar el polinomio, dado un conjunto de puntos y valores. El problema de la evaluación consiste en encontrar los valores, dado el polinomio y los puntos. (El problema de encontrar los puntos, dado el polinomio y los valores, se denomina *búsqueda de raíces*.)

La solución clásica al problema de la interpolación está dada por la fórmula de interpolación de Lagrange, que se utiliza a menudo como prueba de que un polinomio de grado $N - 1$ está completamente determinado por N puntos:

$$p(x) = \sum_{1 \leq j \leq N} y_j \prod_{\substack{1 \leq i \leq N \\ i \neq j}} \frac{x - x_i}{x_j - x_i}.$$

Esta fórmula parece en principio «indescifrable», pero realmente es bastante simple. Por ejemplo, el polinomio de grado 2 tal que $p(1) = 3$, $p(2) = 7$, y $p(3) = 13$ está dado por

$$p(x) = 3 \frac{x - 2}{1 - 2} \frac{x - 3}{2 - 3} + 7 \frac{x - 1}{2 - 1} \frac{x - 3}{2 - 3} + 13 \frac{x - 1}{3 - 1} \frac{x - 2}{3 - 2}$$

que se simplifica a

$$x^2 + x + 1.$$

Para un x que pertenece a x_1, x_2, \dots, x_N , la fórmula se construye de forma que $p(x_k) = y_k$ para $1 < k < N$, dado que el producto es 0 excepto si $j = k$, en el que es 1. En el ejemplo, los dos últimos términos son cero cuando $x = 1$, los términos primero y último son cero cuando $x = 2$, y los dos últimos son cero cuando $x = 3$.

La conversión de un polinomio entre la forma descrita por la fórmula de Lagrange y la representación estándar de los coeficientes no es totalmente directa. Parecen necesitarse al menos N^2 operaciones, ya que hay N términos en la suma que consisten en productos de N factores. Realmente, es preciso tener cierta destreza para obtener un algoritmo cuadrático, ya que los factores no son números exactamente, sino polinomios de grado N . A pesar de ello, cada término es muy similar al anterior. El lector podría estar interesado en descubrir cómo aprovecharse de esto para obtener un algoritmo cuadrático. Este ejercicio

permite apreciar la dificultad de escribir un programa eficaz para llevar a cabo los cálculos implicados en una fórmula matemática.

Cómo en la evaluación polinómica, existen métodos más sofisticados que pueden resolver el problema en $N(\log N)^2$ pasos, y en el Capítulo 41 se verá un método que utiliza sólo $N \log N$ multiplicaciones para un conjunto específico de N puntos de interés.

Multiplicación polinómica

El primer algoritmo aritmético más sofisticado que se va a presentar es para el problema de la *multiplicación polinómica*: dados dos polinomios $p(x)$ y $q(x)$, calcular su producto $p(x)q(x)$. Como se apuntó al principio de este capítulo, los polinomios de grado $N - 1$ pueden tener N términos (incluidas las constantes) y su producto es de grado $2N - 2$ y puede tener hasta $2N - 1$ términos. Por ejemplo,

$$(1 + x + 3x^2 - 4x^3)(1 + 2x - 5x^2 - 3x^3) = (1 + 3x - 6x^3 - 26x^4 - 11x^5 + 12x^6).$$

El sencillo algoritmo para este problema dado al principio de este capítulo necesita N^2 multiplicaciones para polinomios de grado $N - 1$: cada uno de los N términos de $p(x)$ se multiplica por cada uno de los N términos de $q(x)$.

Para mejorar el algoritmo, es preciso «dividir y vencer». Una forma de dividir un polinomio en dos consiste en separar los coeficientes en dos mitades: dado un polinomio de grado $N - 1$ (con N coeficientes), se puede dividir en dos polinomios con $N/2$ coeficientes (suponiendo que N es par): se utilizan los $N/2$ coeficientes de menor orden para un polinomio y los $N/2$ coeficientes de mayor orden para el otro. Para $p(x) = p_0 + p_1x + \dots + p_{N-1}x^{N-1}$, se definen

$$\begin{aligned} p_f(x) &= p_0 + p_1x + \dots + p_{N/2-1}x^{N/2-1}, \\ p_h(x) &= p_{N/2} + p_{N/2+1}x + \dots + p_{N-1}x^{N/2-1}. \end{aligned}$$

A continuación se divide $q(x)$ de la misma forma, y se tiene:

$$\begin{aligned} p(x) &= p_f(x) + x^{N/2}p_h(x), \\ q(x) &= q_f(x) + x^{N/2}q_h(x). \end{aligned}$$

Ahora, en términos de los polinomios más pequeños, el producto está dado por:

$$p(x)q(x) = p_f(x)q_f(x) + (p_f(x)q_h(x) + q_f(x)p_h(x))x^{N/2} + p_h(x)q_h(x)x^N.$$

(Se utiliza la misma partición del Capítulo 35 para evitar desbordamientos.)

Ahora, el enfoque de estas manipulaciones es que sólo se necesitan *tres* multiplicaciones para calcular estos productos (no cuatro, como parecería en la fórmula anterior), porque si se calculan $r_l(x) = p_l(x)q_l(x)$, $r_h(x) = p_h(x)q_h(x)$, y $r_m(x) = (p_l(x) + p_h(x))(q_l(x) + q_h(x))$, se puede obtener el producto $p(x)q(x)$ calculando

$$p(x)q(x) = r_l(x) + (r_m(x) - r_l(x) - r_h(x))x^{N/2} + r_h(x)x^N.$$

La ganancia que se obtiene puede que no sea evidente en este pequeño ejemplo. El método está basado en el hecho de que la suma de polinomios requiere un algoritmo lineal, y que la multiplicación del polinomio de fuerza bruta es cuadrática, así que merece la pena realizar unas pocas adiciones (fáciles) para evitar una multiplicación (difícil). Posteriormente se examinarán con más cuidado los resultados que este método puede proporcionar.

Para el ejemplo anterior, con $p(x) = 1 + x + 3x^2 - 4x^3$ y $q(x) = 1 + 2x - 5x^2 - 3x^3$, se tiene

$$r_l(x) = (1 + x)(1 + 2x) = 1 + 3x + 2x^2,$$

$$r_h(x) = (3 - 4x)(-5 - 3x) = -15 + 11x + 12x^2,$$

$$r_m(x) = (4 - 3x)(-4 - x) = -16 + 8x + 3x^2.$$

Así, $r_m(x) - r_l(x) - r_h(x) = -2 - 6x - 11x^2$, y el producto se calcula como la suma de los tres términos de acuerdo con la fórmula anterior:

$$\begin{aligned} p(x)q(x) &= (1 + 3x + 2x^2) \\ &\quad + (-2 - 6x - 11x^2)x^2 \\ &\quad + (-15 + 11x + 12x^2)x^4 \\ &= 1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6. \end{aligned}$$

Esta estrategia de divide y vencerás resuelve el problema de la multiplicación polinómica de tamaño N resolviendo tres subproblemas de tamaño $N/2$, utilizando algunas sumas polinómicas para organizar los subproblemas y combinar sus soluciones. (Si $N = 1$, el producto es el producto escalar de los dos coeficientes constantes.) Así, este procedimiento se describe fácilmente como un programa recursivo:

```
float *mult(float p[], float q[], int N)
{
    float p1[N/2], q1[N/2], ph[N/2], qh[N/2],
          t1[N/2], t2[N/2];
    float r[2*N-2], r1[N], rm[N], rh[N];
```

```

int i, N2;
if (N == 1)
    { r[0] = p[0]*q[0]; return (float *) r; }
for (i = 0; i < N/2; i++)
    { pl[i] = p[i]; ql[i] = q[i]; }
for (i = N/2; i < N; i++)
    { ph[i-N/2] = p[i]; gh[i-N/2] = q[i]; }
for (i = 0; i < N/2; i++) t1[i] = pl[i]+ph[i];
for (i = 0; i < N/2; i++) t2[i] = ql[i]+qh[i];
rm = mult(t1, t2, N/2);
rl = mult(pl, ql, N/2);
rh = mult(ph, qh, N/2);
for (i = 0; i < N-1; i++) r[i] = rl[i];
r[N-1] = 0;
for (i = 0; i < N-1; i++) r[N+i] = rh[i];
for (i = 0; i < N-1; i++)
    r[N/2+i] += rm[i] - (rl[i]+rh[i]);
return (float *) r;
}

```

Aunque el código anterior es una descripción sucinta de este método, por desgracia no es un programa válido en C++ porque las funciones no pueden declarar dinámicamente arrays. Esto se puede manipular en C++ asignando y liberando memoria explícitamente para el array temporal o representando todos los polinomios por listas enlazadas; esto se deja como ejercicio para el lector. El programa anterior supone que N es una potencia de dos, aunque la generalización para un N cualquiera no es difícil. Las principales dificultades consisten en asegurar que la recursión termina correctamente y que los polinomios se dividen de forma apropiada cuando N es impar.

¿Por qué este método de divide y vencerás constituye una mejora? Para encontrar la respuesta, es preciso resolver una fórmula básica de recurrencia ligeramente más complicada que la del Capítulo 6.

Propiedad 36.1 *Dos polinomios de grado N se pueden multiplicar utilizando $N^{1.58}$ multiplicaciones aproximadamente.*

Según el programa recursivo, está claro que el número de multiplicaciones enteras necesarias para multiplicar dos polinomios de tamaño N es el mismo que para multiplicar tres pares de polinomios de grado $N/2$. (Obsérvese que, por ejemplo, no se necesitan multiplicaciones para calcular $r(x)x^N$, sino movimiento de datos.) Si M_N es el número de multiplicaciones necesarias para el producto de dos polinomios de grado N , se tiene

$$M_N = 3M_{N/2}, \quad \text{para } N \geq 2 \text{ con } M_1 = 1.$$

Así $M(2) = 3$, $M(4) = 9$, $M(8) = 27$, etc. Como en el Capítulo 6, si se toma $N = 2^n$, se puede aplicar repetidamente la recurrencia para encontrar la solución:

$$M(2^n) = 3M(2^{n-1}) = 3^2M(2^{n-2}) = 3^3M(2^{n-3}) = \dots = 3^nM(1) = 3^n.$$

Si $N = 2^n$, entonces $3^n = 2^{(\lg 3)n} = 2^{n\lg 3} = N^{\lg 3}$. Aunque esta solución es exacta sólo para $N = 2^n$, en general se obtiene que

$$M_N \approx N^{\lg 3} \approx N^{1.58},$$

lo cual es un ahorro sustancial con respecto al sencillo método en N^2 . ■

Se observa que si se hubieran utilizado las cuatro multiplicaciones en el método simple de divide y vencerás, se tendría el mismo rendimiento que en el método elemental porque la recurrencia sería $M(N) = 4M(N/2)$, con la solución $M(2^n) = 4^n = N^2$.

Este método ilustra exactamente la técnica de divide y vencerás, pero se utiliza pocas veces en la práctica porque se conoce un método de divide y vencerás mucho mejor, que se estudiará en el Capítulo 41. Dicho método divide el problema original en sólo dos subproblemas, con un pequeño procesamiento extra. Se obtiene así una recurrencia estándar de divide y vencerás $M_N = 2M_{N/2} + N$ para el número de multiplicaciones necesarias y se puede afirmar de M_N que es aproximadamente $N\lg N$.

Operaciones aritméticas sobre enteros grandes

Un gran entero se puede considerar como un polinomio con restricciones en los coeficientes. Por ejemplo, el entero de 28 cifras

0120200103110001200004012314

podría corresponder al polinomio

$$x^{26} + 2x^{25} + 2x^{23} + x^{20} + 3x^{18} + x^{17} + x^{16} + x^{12} + 2x^{11} + 4x^6 + x^4 + 2x^3 + 3x^2 + x + 4.$$

Es decir, el número es el valor del polinomio en $x = 10$. Recíprocamente, todo polinomio de grado $N - 1$ con coeficientes positivos menores que 10 corresponde precisamente a un entero de N cifras.

Así, se pueden utilizar las operaciones polinómicas para manipular grandes

enteros. Dicho de otra forma, se representan los enteros simplemente como arrays, y se utilizan las rutinas de manipulación polinómica desarrolladas, como si los arrays representaran polinomios. Por ejemplo, para multiplicar dos números de 100 cifras, se podría utilizar el algoritmo anterior para calcular un número de 200 cifras. El defecto de esta estrategia es que los coeficientes del resultado probablemente no serán inferiores a 10. Esto se puede remediar en un solo paso: empezando en $i=0$, añadiendo $p[i]/10$ a $p[i+1]$, sustituyendo $p[i]$ por $p[i] \% 10$ e incrementando i , continuando hasta que no queden coeficientes.

Se podría utilizar una base mayor de 10 cifras; por ejemplo, el anterior número de 28 cifras podría corresponder también al polinomio

$$120x^6 + 2001x^5 + 311x^4 + x^3 + 2000x^2 + 401x + 2314.$$

Evaluando este polinomio en $x=10.000$ se obtiene el entero. Así se pueden representar enteros con menos memoria (una cuarta parte menos en este ejemplo), pero exponiéndose a la posibilidad de desbordamiento en los coeficientes durante alguna operación intermedia. Son bastante conocidas las restricciones matemáticas a utilizar con bases grandes pero, en la práctica, es mejor ser conservador y utilizar bases pequeñas.

Para el sistema de cripto RSA (Capítulo 23) se debe no sólo multiplicar grandes enteros, sino también exponentiar y dividir. Concretamente, se necesita calcular $M^p \bmod N$ cuando M , p y N son enteros grandes. Esta operación es bastante complicada, pero se puede esbozar el método. En primer lugar, la exponentiación se puede hacer con multiplicaciones sucesivas, como se describió anteriormente, así que es suficiente encontrar la forma de calcular $M_1 M_2 \bmod N$ cuando M_1 , M_2 , y N son enteros grandes. La clave para ejecutar el cálculo del módulo es evaluar $10^i \bmod N$ para todos los 10^i inferiores al mayor entero que se haya encontrado. Así, cualquier módulo particular será una combinación lineal de estos valores. Una base mayor reducirá el número de cálculos necesarios. En términos matemáticos este método corresponde, por ejemplo, a calcular

$$0120200103110001200004012314 \bmod N$$

evaluando

$$\begin{aligned} & 120(x^6 \bmod N) + 2001(x^5 \bmod N) + 311(x^4 \bmod N) \\ & + (x^3 \bmod N) + 2000(x^2 \bmod N) + 401(x \bmod N) + 2314 \end{aligned}$$

para $x = 10.000$. Los valores $10.000^i \bmod N$ se pueden calcular con anterioridad y almacenarse en una tabla, o calcular de forma incremental cuando sea necesario, como en el algoritmo de búsqueda de cadenas de Rabin-Karp del Capítulo 19.

Aritmética de matrices

La implementación de operaciones básicas sobre las matrices implica consideraciones similares a las presentadas anteriormente para los polinomios. Por ejemplo, la suma de dos matrices es trivial porque es una suma término a término, como para los polinomios.

La multiplicación de matrices es también directa. Si r es el producto de p y q , el elemento $r[i][j]$ es el *producto escalar* de la i -ésima fila de p por la j -ésima columna de q . El producto escalar es simplemente la suma de las N multiplicaciones término a término

$$p_{i1}q_{1j} + p_{i2}q_{2j} + \dots + p_{i(N-1)}q_{(N-1)j},$$

como en el siguiente programa:

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0, r[i][j] = 0; k < N; k++)
            r[i][j] += p[i][k]*q[k][j];
```

Es posible que el lector desee utilizar el siguiente ejemplo para examinar el código anterior:

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} 8 & 3 & 0 \\ 3 & 10 & 2 \\ 0 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 17 & 25 & -18 \\ 11 & 9 & -10 \\ -14 & -13 & 26 \end{pmatrix}.$$

Cada uno de los N^2 elementos de la matriz producto se calcula con N multiplicaciones, así que se necesitan aproximadamente N^3 operaciones para multiplicar dos matrices $N * N$.

Como en los polinomios, las matrices *dispersas* (aquellas que tienen muchos elementos nulos) se pueden procesar de forma mucho más eficaz utilizando una representación por lista enlazada. Para conservar la estructura bidimensional intacta, cada elemento no cero de la matriz se representa con un nodo de la lista, que contiene un valor y dos enlaces, uno apuntando al siguiente elemento no cero de la misma fila y el otro al siguiente elemento no cero en la misma columna. La implementación de la suma de matrices dispersas representadas de esta forma es similar a la implementación de polinomios dispersos, pero más complicada por el hecho de que cada nodo aparece en dos listas.

La aplicación más famosa de la técnica de divide y vencerás a un problema aritmético es el método de Strassen para la multiplicación de matrices. No se

darán aquí los detalles, pero se puede esbozar un método muy similar en concepto al de multiplicación de polinomios que se acaba de presentar.

El método directo para la multiplicación de dos matrices $N * N$ necesita N^3 multiplicaciones escalares, puesto que cada uno de los N^2 elementos de la matriz producto se obtiene de N multiplicaciones. El método de Strassen sirve para dividir el problema en dos, lo que corresponde a dividir las matrices en cuatro, cada una de $N/2 * N/2$. El problema que resulta es equivalente a multiplicar matrices $2 * 2$. Del mismo modo que se pudo reducir el número de multiplicaciones necesarias de cuatro a tres por combinación de términos en el problema de la multiplicación de polinomios, Strassen fue capaz de encontrar un método para combinar los términos y reducir el número de multiplicaciones necesarias para el problema de la multiplicación de matrices $2 * 2$ de 8 a 7. La reordenación y los términos necesarios son bastante complicados.

Propiedad 36.2 *Dos matrices N por N se pueden multiplicar utilizando aproximadamente $N^{2.81}$ multiplicaciones.*

Del análisis anterior se obtiene que el número de multiplicaciones necesarias para la multiplicación de matrices utilizando el método de Strassen está definido por la recurrencia de divide y vencerás $M(N) = 7M(N/2)$, que tiene la solución $M(N) \approx N^{1.87} \approx N^{2.81}$, como se vio antes. ■

Este resultado fue bastante sorprendente cuando apareció en 1968, puesto que se había pensado previamente que eran necesarias N^3 multiplicaciones para la multiplicación de matrices. El problema ha sido estudiado muy intensamente durante los últimos años, y se han encontrado métodos ligeramente superiores al de Strassen. Todavía no se ha encontrado el «mejor» algoritmo para la multiplicación de matrices, y esto constituye uno de los más famosos problemas pendientes de la informática.

Es importante señalar que se han considerado solamente multiplicaciones. Antes de elegir un algoritmo para una aplicación práctica, se deben considerar también los costes de las sumas y restas extras relativos a la combinación de los términos y los de las llamadas recursivas. Estos costes pueden depender considerablemente de la implementación particular o de la computadora utilizada. Pero dichos sobrecostes hacen que el método de Strassen claramente sea menos eficaz que el método estándar para matrices pequeñas. Incluso para grandes matrices, en términos del número de elementos de los datos de entrada, el método de Strassen representa realmente una mejora sólo de $N^{1.5}$ a $N^{1.41}$. Esta mejora es difícil de notar excepto para N muy grandes. Por ejemplo, N debería ser superior a un millón para que el método de Strassen efectúe como mucho un cuarto de las multiplicaciones del método estándar, incluso aunque el sobrecoste por multiplicación sea cuatro veces mayor. De este modo el algoritmo es una contribución teórica, no práctica.

Ejercicios

1. Los polinomios se pueden representar de la forma $r_0(x - r_1)(x - r_2) \dots (x - r_N)$. ¿Cómo se podrían multiplicar dos polinomios en esta representación?
2. Escribir un programa en C++ para multiplicar dos polinomios dispersos, utilizando una representación por lista enlazada que excluya los nodos de los términos que tengan coeficientes nulos.
3. Escribir un programa en C++ que coloque el valor v en el elemento de la fila i y columna j de una matriz dispersa suponiendo que la matriz se representa con una lista enlazada que no tiene nodos en las entradas nulas.
4. Obtener un método para evaluar un polinomio de raíces conocidas r_1, r_2, \dots, r_N , y comparar dicho método con el de Horner.
5. Escribir un programa para evaluar polinomios utilizando el método de Horner, cuando los polinomios están representados por listas enlazadas. Asegurarse de que el programa funciona eficazmente sobre polinomios dispersos.
6. Escribir un programa N^2 para efectuar una interpolación de Lagrange.
7. ¿Se puede calcular x^{55} con menos de nueve multiplicaciones? En caso afirmativo, decir cómo, y si no, decir por qué no.
8. Listar todas las multiplicaciones polinómicas efectuadas si el método de multiplicación de polinomios, del tipo divide y vencerás, `mult`, descrito en el texto, se utiliza para elevar al cuadrado el polinomio $1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8$.
9. El método `mult` se podría hacer más eficaz para polinomios dispersos haciendo que devuelva cero si todos los coeficientes de uno de sus elementos son cero. ¿Aproximadamente, cuántas multiplicaciones (con un factor constante) efectuará un tal programa para elevar al cuadrado $1 + x^N$?
10. Implementar una versión de trabajo de `mult` que utilice una representación por lista enlazada y determinar de forma empírica un valor de N para el que sea más rápida que el método de fuerza bruta (que utilice la misma representación).

Eliminación gaussiana

Uno de los cálculos científicos más importantes es la resolución de sistemas de ecuaciones simultáneas. El algoritmo fundamental para resolver dichos sistemas, la *eliminación gaussiana*, es relativamente simple y ha cambiado poco desde que fue inventado hace 150 años. Este algoritmo es bastante bien conocido en la actualidad, sobre todo en el curso de los últimos 20 años, por lo que se puede utilizar con toda la seguridad de que proporcionará eficazmente resultados exactos.

La eliminación de Gauss es un ejemplo de algoritmo que seguramente estará disponible en la mayoría de los sistemas informáticos, y de hecho es una primitiva en ciertos lenguajes como APL o Basic. Sin embargo, como el algoritmo básico es fácil de comprender e implementar, y en algunos casos particulares se desea disponer de variantes del algoritmo mejores que la versión estándar que se proporciona en la biblioteca, el método merece ser estudiado como uno de los métodos numéricos más importantes de los utilizados hoy en día.

Como en otros aspectos matemáticos ya estudiados anteriormente, el tratamiento resaltará solamente los principios básicos y será autosuficiente. No se necesita tener conocimientos de álgebra lineal para comprender el método básico. Se desarrollará una sencilla implementación en C++, que será tan fácil de utilizar para aplicaciones simples como una subrutina de biblioteca. Sin embargo, también se examinarán ejemplos de mayor dificultad, susceptibles de aparecer en la realidad. Sin duda alguna, para las aplicaciones grandes o importantes, será mejor recurrir a una implementación más sintonizada, así como tener alguna familiaridad con las matemáticas subyacentes.

Un ejemplo simple

Supóngase que se tienen tres variables, x , y y z y las tres ecuaciones siguientes:

$$x + 3y - 4z = 8,$$

$$x + y - 2z = 2,$$

$$-x - 2y + 5z = -1.$$

El objetivo de este capítulo será calcular los valores de las variables que satisfagan simultáneamente a las ecuaciones. Según sean éstas puede que el problema no tenga ninguna solución (por ejemplo, si dos ecuaciones, como $x + y = 1$ y $x + y = 2$, son incompatibles) o varias soluciones (por ejemplo, cuando dos ecuaciones son idénticas o cuando hay más variables que ecuaciones). Se supone que el número de variables y de ecuaciones es el mismo, y se estudiará un algoritmo que encuentre una solución única, si es que existe.

Para facilitar la generalización de las fórmulas a más de tres puntos, se comenzará por renombrar las variables por medio de subíndices:

$$x_1 + 3x_2 - 4x_3 = 8,$$

$$x_1 + x_2 - 2x_3 = 2,$$

$$-x_1 - 2x_2 + 5x_3 = -1.$$

Para evitar tener que estar escribiendo continuamente las variables, es conveniente utilizar la notación matricial para describir el sistema. Las anteriores ecuaciones son exactamente equivalentes a la ecuación matricial

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}.$$

Existen varias operaciones que se pueden aplicar a estas ecuaciones sin que cambie la solución:

Intercambiar ecuaciones: Evidentemente, el orden en el que se escriben las ecuaciones no influye sobre la solución. En la representación matricial esta operación corresponde a intercambiar dos líneas de la matriz (y del vector columna del miembro de la derecha).

Renombrar variables: Esto corresponde a intercambiar columnas en la representación matricial (si se intercambian las columnas i y j también deben intercambiarse las variables x_i y x_j .)

Multiplicar ecuaciones por una constante: Una vez más, en la representación matricial esto corresponde a multiplicar una fila de la matriz (y el elemento correspondiente del vector columna del miembro de la derecha) por una constante.

Sumar dos ecuaciones y reemplazar una de ellas por esta suma.

No supone mucho esfuerzo el convencerse uno mismo de que estas operaciones, especialmente la última, no afectan a la solución. Por ejemplo, se obtiene un sistema de ecuaciones equivalente a uno de los anteriores reemplazando la segunda ecuación por la diferencia entre las dos primeras:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ -1 \end{pmatrix}.$$

Obsérvese que esta transformación ha eliminado la variable x_1 de la segunda ecuación. De igual forma se puede eliminar esta variable de la tercera ecuación reemplazando a ésta por la suma de la primera y la tercera:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ 7 \end{pmatrix}.$$

Ahora se ha eliminado la variable x_1 de todas las ecuaciones menos de la primera. Se puede así transformar sistemáticamente el sistema original en otro que tenga la misma solución, pero que sea más fácil de resolver. En este ejemplo, sólo es necesario un paso más, que combina dos de las operaciones anteriores: reemplazar la tercera ecuación por la diferencia entre la segunda y dos veces la tercera. Esto hace que todos los elementos situados por debajo de la diagonal principal sean ceros, quedando un sistema de ecuaciones que es particularmente fácil de resolver. El sistema del ejemplo queda de la siguiente forma:

$$x_1 + 3x_2 - 4x_3 = 8,$$

$$2x_2 - 2x_3 = 6,$$

$$-4x_3 = -8.$$

Ahora se puede resolver directamente la tercera ecuación: $x_3 = 2$. Si se sustituye este valor en la segunda ecuación se puede calcular el de x_2 :

$$2x_2 - 4 = 6,$$

$$x_2 = 5.$$

De forma similar, sustituyendo estos dos valores en la primera ecuación se puede calcular el de x_1 :

$$x_1 - 15 - 8 = 8,$$

$$x_1 = 1.$$

lo que completa la resolución del sistema.

Este ejemplo ilustra las dos fases básicas de la eliminación gaussiana. La primera es la fase de *eliminación descendente*, durante la que se transforma el sistema original, por medio de la eliminación sistemática de variables en las ecuaciones, en un sistema que sólo contiene ceros por debajo de la diagonal principal. Este proceso se denomina algunas veces *triangulación*. La segunda fase se denomina *fase de sustitución ascendente*, en la que los valores de las variables se calculan por medio de la matriz triangular obtenida en la primera fase.

Esbozo del método

En general, se desea resolver un sistema de N ecuaciones con N incógnitas:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1N}x_N = b_1,$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2N}x_N = b_2,$$

.

.

.

$$a_{N1}x_1 + a_{N2}x_2 + \dots + a_{NN}x_N = b_N.$$

Estas ecuaciones se escriben en forma matricial como una ecuación única:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & & & \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

o simplemente $Ax = b$, donde A representa la matriz, x las variables y b los miembros de la derecha de las ecuaciones. Como las líneas de A se tratan según los elementos de b es conveniente considerar a b como la $(N + 1)$ -ésima columna de A y utilizar un array $N^* (N + 1)$ para almacenar a los dos.

La fase de eliminación descendente se puede resumir como sigue: se comienza por eliminar la primera variable de todas las ecuaciones excepto la primera, añadiendo un múltiplo apropiado de la primera ecuación a cada una de ellas, eliminando a continuación la segunda variable de todas las ecuaciones a partir de la segunda, añadiendo un múltiplo apropiado de esta segunda a cada una de las otras; después se elimina la tercera variable de todas las ecuaciones a partir de la tercera, etc. Para eliminar la i -ésima variable de la j -ésima ecuación (para j entre $i + 1$ y N) se multiplica la i -ésima ecuación por a_{ij}/a_{ii} y se resta el resultado de la j -ésima ecuación. Este proceso se describe más sucintamente por medio del siguiente fragmento de código:

```
for (i = 1; i <= N; i++)
    for (j = i+1; j <= N; j++)
        for (k = N+1; k >= i; k--)
            a[j][k] -= a[i][k]*a[j][i]/a[i][i];
```

El código consiste en tres bucles anidados y el tiempo total de ejecución es en esencia proporcional a N^3 . El tercer bucle procede hacia atrás para evitar la destrucción de $a[j][i]$ antes de que se utilice para ajustar los valores de los otros elementos de la misma fila.

El fragmento de código del párrafo anterior es demasiado simple para que siempre funcione bien: $a[i][i]$ podría ser cero, y por ello pudiera ocurrir una división por cero. Sin embargo, esto es fácil de remediar dado que se puede cambiar cualquier fila (entre $i + 1$ y N) con la i -ésima fila para hacer $a[i][i]$ distinto de cero en el otro bucle. Si no se puede encontrar ninguna fila para intercambiar entonces la matriz es *singular*: el sistema no tiene solución única. (El programa podría informar de esto explícitamente o sería posible dejar que el error apareciese como último recurso como una división por cero.) Se necesitará añadir algunas instrucciones al fragmento de código anterior para permitir la búsqueda de una fila inferior con un elemento distinto de cero en la i -ésima columna y después intercambiar esta fila con la i -ésima fila. El elemento $a[i][i]$ que se utiliza finalmente para eliminar los elementos distintos de cero que están por debajo de la diagonal principal en la i -ésima columna se denomina el *pivote*.

De hecho, es recomendable hacer algo más que encontrar una fila que contenga un valor no nulo en la i -ésima columna. Es preferible utilizar la fila (entre $i + 1$ y N) cuyo valor de la i -ésima columna es el mayor en valor absoluto. La razón es que pueden aparecer graves errores de cálculo si el valor de pivote utilizado como «escala» de una línea es demasiado pequeño. Si $a[i][i]$ es muy pequeño entonces el factor de escalación $a[j][i]/a[i][i]$ utilizado para eliminar la i -ésima variable de la j -ésima ecuación (para j entre $i + 1$ y N) será muy grande. De hecho, puede ser tan grande como para empequeñecer los coeficientes reales $a[j][k]$ hasta el punto de que el valor de $a[j][k]$ se distorsione por «errores de redondeo».

De forma más simple, los números que difieren en magnitudes muy grandes

no se pueden sumar ni restar en el sistema de números de coma flotante utilizado normalmente para representar a los números reales, y la utilización de un pequeño pivote incrementa la probabilidad de que se lleve a cabo este tipo de operaciones. La utilización de valores mayores en la i -ésima columna entre las filas $i + 1$ y N asegurará que el factor de escalación sea siempre inferior a 1 y permitirá prevenir este tipo de error. Se podría contemplar el hecho de tratar columnas superiores a la i -ésima para encontrar un gran elemento, pero se ha demostrado que se pueden obtener respuestas precisas sin recurrir a estas complicaciones extra.

El código siguiente para la fase de eliminación descendente de la eliminación de Gauss es una implementación directa de este proceso. Para cada i entre 1 y N , se explora hacia abajo la i -ésima columna para encontrar el elemento mayor (en las filas superiores a la i -ésima). La fila que contiene a ese elemento se intercambia con la i -ésima, y a continuación se elimina la variable i de las ecuaciones $i + 1$ a la N exactamente como antes:

```
eliminacion()
{
    int i, j, k, max;
    float t;
    for (i = 1; i <= N; i++)
    {
        max = i;
        for (j = i + 1; j <= N; j++)
            if (abs(a[j][i]) > abs(a[max][i])) max = j;
        for (k = 1; k <= N + 1; k++)
        {
            t = a[i][k];
            a[i][k] = a[max][k];
            a[max][k] = t;
        }
        for (j = i + 1; j <= N; j++)
            for (k = N + 1; k >= i; k--)
                a[j][k] -= a[i][k]*a[j][i]/a[i][i];
    }
}
```

En algunos algoritmos se necesita que el pivote $a[i][i]$ se utilice para eliminar la i -ésima variable de todas las ecuaciones menos la i -ésima (no solamente de las que están entre la $(i + 1)$ -ésima y la N -ésima). Este proceso se denomina de *pivotado total*; en la eliminación descendente solamente se hace parte de este trabajo, y por eso se denomina el proceso como de *pivotado parcial*. En este libro se examinará el algoritmo de pivotado total en el Capítulo 43.

Después de completar la fase de eliminación descendente, el array a tiene todos los elementos situados por debajo de la diagonal principal iguales a cero

y se puede ejecutar la fase de sustitución ascendente. El programa para ello es todavía más directo:

```
sustitucion()
{
    int j, k;
    float t;
    for (j = N; j >= 1; j--)
    {
        t = 0.0;
        for (k = j + 1; k <= N; k++) t + a[j][k]*x[k];
        x[j] = (a[j][N+1]-t)/a[j][j];
    }
}
```

Una llamada a `eliminacion` seguida por una llamada a `sustitucion` calcula la solución en el array `x` de `N` elementos. La división por cero podría producirse todavía en matrices singulares —una rutina de «biblioteca» comprobaría este caso de forma explícita—. De hecho, la mayor parte de las rutinas de biblioteca efectúan comprobaciones mucho más amplias, como se acaba de ver.

Propiedad 37.1 *Un sistema de N ecuaciones simultáneas con N incógnitas puede resolverse utilizando aproximadamente $N^3/3$ multiplicaciones y sumas.*

El tiempo de ejecución de `sustitucion` está en $O(N^2)$, por lo que la mayor parte del trabajo está hecho en `eliminacion`. El análisis de esta rutina muestra que para cada valor de i el bucle k se repite $N - i + 2$ veces y el bucle j $N - i$ veces; esto significa que el bucle interno se ejecuta $\sum_{1 \leq i \leq N} N(N - i + 2)(N - i) = N^3/3 + O(N^2)$ veces. El valor de $-a[j][i]/a[i][i]$ se puede calcular fuera del bucle k , y así el bucle interno consiste en una multiplicación y una suma.■

Una forma alternativa de proceder, después de que la eliminación descendente ha creado todos los ceros por debajo de la diagonal principal, es utilizar exactamente el mismo método para poner ceros en todos los elementos que están situados por encima de dicha diagonal: primero se pone a cero la última columna, menos $a[N][N]$, añadiendo el múltiplo adecuado de $a[N][N]$, y después se hace lo mismo con la penúltima columna, etc. Esto es, se hace otra vez un «pivotado parcial», pero en la otra «parte» de cada columna, trabajando hacia atrás a lo largo de ellas. Después de que se termine este proceso (denominado *reducción de Gauss-Jordan*), solamente son distintos de cero los elementos de la diagonal principal, lo que proporciona una solución trivial. Sin embargo, el número de operaciones aritméticas utilizadas en este proceso es muy superior al de la sustitución ascendente.

Los errores de cálculo son el principal motivo de preocupación en lo que respecta a la eliminación de Gauss. Como se mencionó anteriormente, es preciso ser conscientes de las situaciones en las que las magnitudes de los coeficientes difieren fuertemente. La utilización del mayor elemento disponible en una columna para el pivotado parcial asegura que no se crearán arbitrariamente coeficientes mayores en el proceso de pivotado, pero no siempre es posible evitar los errores graves. Por ejemplo, aparecen coeficientes muy pequeños cuando dos ecuaciones diferentes tienen coeficientes muy próximos entre sí. Sin embargo, realmente es posible determinar con antelación si existe el riesgo de que se introduzcan estos problemas en la solución. Se puede asociar a cada matriz un valor numérico denominado *número de condición*, que puede utilizarse para estimar la exactitud de la respuesta calculada. Una buena subrutina de biblioteca para la eliminación de Gauss calculará el número de condición de la matriz a la vez que la solución, permitiendo así conocer la precisión de esta última. El tratamiento completo de los temas tratados aquí supera el alcance de este libro.

El empleo del método de eliminación de Gauss con pivotado parcial sobre el mayor pivote disponible «garantiza» que produce resultados con errores de cálculo muy pequeños. Existen resultados matemáticos muy «trabajados» que permiten demostrar que la solución calculada es muy precisa, excepto para el caso de matrices degeneradas (que pueden ser más indicativas de problemas en el sistema de ecuaciones que en el método de resolución). El algoritmo ha sido objeto de estudios teóricos muy detallados, y se puede recomendar como procedimiento de cálculo de aplicación muy amplia.

Variaciones y extensiones

El método que se acaba de presentar es el más apropiado para matrices $N \times N$ en las que la mayor parte de los elementos no sean ceros. Como se ha visto para otros problemas, existen técnicas especiales más adecuadas para las matrices *dispersas* en las que la mayoría de los elementos son ceros. Esta situación corresponde a los sistemas de ecuaciones en los que cada una de las ecuaciones tiene solamente algunos términos.

Si los elementos distintos de cero no tienen una estructura determinada, se recomienda la representación por listas enlazadas presentada en el Capítulo 36, con un nodo por cada elemento de la matriz distinto de cero, enlazado con los adyacentes de la fila y columna. El método estándar se puede implementar para esta representación, pero normalmente con complicaciones extra debido a la necesidad de crear y destruir los elementos distintos de cero. Esta técnica presenta pocas posibilidades de tener ventajas si sólo se dispone de memoria para contener toda la matriz, siendo además mucho más compleja que el método estándar. Además, las matrices dispersas se hacen sensiblemente más dispersas durante el proceso de la eliminación de Gauss.

En algunas matrices no sólo se tienen pocos elementos distintos de cero, sino que presentan también una estructura simple, por lo que no son necesarias las listas enlazadas. El ejemplo más común de esto es la matriz «banda», en la que los elementos no nulos están todos situados en las cercanías de la diagonal. En estos casos, el bucle interno de los algoritmos de la eliminación de Gauss sólo necesita repetirse unas pocas veces, por lo que el tiempo total de ejecución (y las necesidades de almacenamiento) es proporcional a N , no a N^3 .

Un caso particular interesante de la matriz de banda es la matriz «tridiagonal» en la que sólo son distintos de cero los elementos situados directamente encima, debajo o sobre la diagonal. Por ejemplo, la forma general de una matriz tridiagonal para $N = 5$ es:

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{pmatrix}$$

Para estas matrices, las fases de eliminación descendente y sustitución ascendente se reducen todas a un bucle `for` sencillo:

```
for (i = 1; i < N; i++)
{
    a[i+1][N+1] -= a[i][N+1]*a[i+1][i]/a[i][i];
    a[i+1][i+1] -= a[i][i+1]*a[i+1][i]/a[i][i];
}
for (j = N; j >= 1; j--)
    x[j] = (a[j][N+1]-a[j][j+1]*x[j+1])/a[j][j];
```

Para la eliminación descendente sólo se necesita considerar los casos de $j = i + 1$ y $k = i + 1$, dado que $a[i][k]$ es 0 para k mayor que $i + 1$. (El caso $k = i$ se puede saltar, ya que coloca un 0 en un elemento del array que no se volverá a examinar otra vez; se podría hacer este cambio en el método simple de la eliminación de Gauss).

Propiedad 37.2 *Un sistema tridiagonal de ecuaciones simultáneas se puede resolver en tiempo lineal.*

Por supuesto, no se necesita utilizar un array bidimensional de tamaño N^2 para representar una matriz tridiagonal. La memoria que se necesita para el programa anterior se puede hacer lineal en N , manteniendo cuatro arrays en lugar de la matriz a : uno para cada una de las tres diagonales distintas de cero y otro para la $(N + 1)$ -ésima columna. Obsérvese que este programa no toma como

pivote necesariamente al mayor de los elementos disponibles, por lo que no se puede prevenir con seguridad la división por cero o la acumulación de errores de cálculo. Sin embargo, para algunos tipos de matrices tridiagonales que aparecen frecuentemente se puede demostrar que esto no es una razón de importancia.■

La reducción de Gauss-Jordan se puede implementar con pivotado total para reemplazar a una matriz por su *inversa*. La inversa de una matriz A , que se representa por A^{-1} , tiene la propiedad de que el sistema de ecuaciones $Ax = b$ se puede resolver por medio de la sencilla multiplicación de matrices $x = A^{-1}b$. Todavía se necesitan N^3 operaciones para calcular x conociendo b . Sin embargo, existe una forma de preprocesar una matriz y «descomponerla» en sus componentes constitutivos, que hace posible la resolución del correspondiente sistema de ecuaciones de cualquier miembro de la derecha en tiempo proporcional a N^2 , con una ganancia de tiempo de un factor N con respecto a la de la utilización del método de eliminación de Gauss. En principio, esto supone memorizar las operaciones realizadas en la $(N + 1)$ -ésima columna durante la fase de eliminación descendente, por lo que el resultado de esta eliminación en una nueva $(N + 1)$ -ésima columna puede calcularse de forma eficaz y realizar la fase de sustitución ascendente de la forma habitual.

Se ha demostrado que la resolución de sistemas de ecuaciones lineales es, desde el punto de vista del cálculo, equivalente a la multiplicación de matrices; así que existen algoritmos (por ejemplo, el de la multiplicación de matrices de Strassen) que pueden resolver sistemas de N ecuaciones con N variables en tiempo proporcional a $N^{2.81\dots}$. Como en el caso de la multiplicación de matrices, el empleo de un método de este tipo no presenta ventajas a menos que se procesen normalmente grandes sistemas de ecuaciones (si es el caso). Como antes, el tiempo de ejecución real de la eliminación de Gauss en función del número de datos de entrada es $N^{3/2}$, lo que es difícil de superar en la práctica.

Ejercicios

1. Obtener la matriz generada por la fase de eliminación descendente de la eliminación de Gauss (*eliminacion*) cuando se utiliza para resolver las ecuaciones $x + y + z = 6$, $2x + y + 3z = 12$ y $3x + y + 3z = 14$.
2. Presentar un sistema de tres ecuaciones con tres incógnitas para el que la implementación sencilla de los tres bucles *for* de la eliminación descendente falle, incluso aunque exista solución.
3. ¿Cuáles son las necesidades de memoria para una eliminación de Gauss en una matriz $N*N$ con sólo $3N$ elementos distintos de cero?
4. Describir lo que sucede cuando se utiliza *eliminacion* en una matriz que tiene una fila con todos sus elementos nulos.

5. Describir lo que sucede cuando se utilizan eliminación y sustitución en una matriz que tiene una columna con todos sus elementos nulos.
6. ¿Cuántas operaciones aritméticas se necesitan aproximadamente para la reducción de Gauss-Jordan?
7. ¿Cuál es el efecto de intercambiar las columnas de una matriz sobre las correspondientes ecuaciones simultáneas?
8. ¿Cómo se podría comprobar la existencia de ecuaciones incompatibles al utilizar la eliminación? Lo mismo para ecuaciones idénticas.
9. ¿Cuál podría ser la utilidad del método de eliminación de Gauss en un sistema de M ecuaciones con N incógnitas con $M < N$? ¿Cuál con $M > N$?
10. Presentar un ejemplo que muestre la necesidad de pivotado del elemento de mayor valor posible, utilizando una computadora imaginaria en la que se pueden representar los números solamente con dos cifras significativas (todos los números deben ser de la forma $x,y \times 10^z$ con x, y y z enteros de una sola cifra).

Ajuste de curvas

El término *ajuste de curvas* (o *ajuste de datos*) se utiliza para describir el problema general de encontrar una función que toma un conjunto de valores dados en un conjunto de puntos también dados. Concretamente, dados los puntos

$$x_1, x_2, \dots, x_N$$

y los valores correspondientes

$$y_1, y_2, \dots, y_N$$

el objetivo consiste en encontrar una función (eventualmente de un determinado tipo) tal que

$$f(x_1) = y_1, f(x_2) = y_2, \dots, f(x_N) = y_N$$

y que además sea tal que $f(x)$ tome valores «razonables» en otros puntos de datos. Podría ser que las x y las y estuvieran relacionadas por una función desconocida y que el objetivo sea encontrar esta función, pero, en general, la definición de «razonable» depende de la aplicación. Se verá que a menudo es fácil identificar las funciones «no razonables».

El ajuste de curvas tiene evidentes aplicaciones en el análisis de datos experimentales así como en otras muchos dominios. Por ejemplo, se puede utilizar en computadoras gráficas para producir curvas de «buena precisión» sin tener que almacenar un gran número de puntos a dibujar. Una aplicación relacionada es la utilización del ajuste de curvas para obtener un algoritmo de cálculo rápido del valor de una función conocida en un punto arbitrario: se memoriza una pequeña tabla de valores exactos y se ajusta la curva para encontrar el resto de valores.

Para enfrentarse a este problema se utilizan dos métodos principales. El primero es la *interpolación*: se busca una función continua que tome exactamente

los valores dados en los puntos dados. El segundo método, *ajuste por mínimos cuadrados*, se utiliza cuando los valores dados no son muy exactos y se busca una función que concuerde con dichos valores lo mejor posible.

Interpolación polinómica

Se ha visto ya un método de resolución de problemas de ajuste de datos: si se sabe que f es un polinomio de grado $N - 1$, se tiene el problema de la interpolación polinómica del Capítulo 36. Incluso si no se tiene un conocimiento particular sobre f , se podría resolver el problema del ajuste de datos aproximando $f(x)$ por interpolación polinómica de grado $N - 1$ para los puntos y valores dados. Esto se podría calcular utilizando los métodos del Capítulo 36, pero existen muchas razones para no utilizar la interpolación polinómica para el ajuste de datos. En principio, la gran cantidad de cálculos implicados (hay disponibles métodos avanzados en $N(\log N)^2$, pero las técnicas elementales son cuadráticas). El cálculo de un polinomio de grado 100 (por ejemplo) parece una «monstruosidad» para interpolar una curva a través de 100 puntos.

El defecto principal de la interpolación polinómica es que los polinomios de grados superiores son funciones relativamente complejas que pueden tener propiedades inesperadas que no se parezcan a las de las funciones que se están ajustando. Un resultado de las matemáticas clásicas (el teorema de aproximación de Weierstrass) dice que es posible aproximar cualquier función razonable por medio de un polinomio (de grado suficientemente elevado). Por desgracia los polinomios de grado muy elevado tienden a fluctuar drásticamente. Así sucede que, aunque se puedan aproximar con bastante precisión la mayoría de las funciones casi en cualquier parte de un intervalo cerrado por medio de una interpolación polinómica, existen siempre algunos lugares donde la aproximación es pésima. Además, esta teoría supone que los valores de los datos son valores exactos tomados por alguna función desconocida, mientras que es habitual el caso de que los valores dados sean sólo aproximados. Si las y son valores aproximados tomados de un polinomio desconocido de grado no muy grande, se puede esperar que los coeficientes de los términos de grado más elevado en la interpolación polinómica sean todos cero. Este caso no es común; en su lugar, el polinomio interpolado intenta utilizar los términos de grado superior para ayudar a alcanzar una mayor exactitud. Estos fenómenos hacen a la interpolación polinómica inapropiada para muchas aplicaciones de ajuste de curvas.

Interpolación spline

No obstante, los polinomios de grado bajo son curvas simples que es fácil describir analíticamente y que tienen una amplia utilización en el ajuste de curvas.

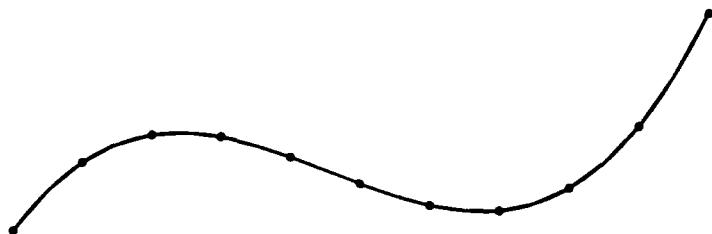


Figura 38.1 Una spline a través de diez nudos.

La astucia consiste en abandonar la idea de intentar hacer que *un* polinomio pase a través de todos los puntos y en su lugar utilizar diferentes polinomios para conectar los puntos adyacentes, colocándolos juntos de forma «continua». Un caso particular, muy elegante, es el denominado *interpolación spline* que también implica cálculos relativamente directos.

Una spline es un dispositivo mecánico utilizado por los dibujantes para dibujar curvas estéticamente agradables: el dibujante fija un conjunto de puntos (*nudos*) en su dibujo, y después deforma una banda de plástico o de madera (la *spline*) en torno a todos ellos y la delinean para producir la curva. La interpolación spline es el equivalente matemático de este proceso y produce la misma curva. La Figura 38.1 muestra una spline a través de diez nudos.

Se puede demostrar por mecánica elemental que la forma tomada por la spline entre dos nudos adyacentes es un polinomio de tercer grado (cúbico). Traducido al problema del ajuste de datos esto significa que se debe considerar la curva como representada por $N - 1$ polinomios cúbicos distintos

$$s_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i, \quad i = 1, 2, \dots, N-1,$$

donde $s_i(x)$ define al polinomio cúbico a utilizar en el intervalo entre x_i y x_{i+1} .

La spline se puede representar de forma evidente por medio de cuatro arrays de una dimensión (o un array $4 \times (N - 1)$ de dos dimensiones). La creación de una spline consiste en calcular los coeficientes a, b, c, d necesarios a partir de los x puntos e y valores dados. Las restricciones físicas de la spline corresponden a las ecuaciones simultáneas que se deben resolver para obtener los coeficientes.

Por ejemplo, evidentemente se debe tener $s_i(x_i) = y_i$ y $s_i(x_{i+1}) = y_{i+1}$ para $i = 1, 2, \dots, N-1$, dado que la spline debe tocar todos los nudos. Además, no sólo debe tocarlos a todos, sino que también debe curvarse suavemente alrededor de ellos sin picos o saltos. Matemáticamente, esto significa que las primeras derivadas de los polinomios de la spline deben ser iguales en los nudos ($s'_{i-1}(x_i) = s'_i(x_i)$ para $i = 2, 3, \dots, N - 1$). De hecho, es preciso que las derivadas segundas de los polinomios deben ser también iguales en los nudos. Estas condiciones dan un total de $4N - 6$ ecuaciones con $4(N - 1)$ coeficientes desconocidos. Se necesitan dos condiciones más para describir la situación de los puntos extremos de

la spline. Existen varias opciones para ello: aquí se utilizará la denominada «natural» que deriva de $s''_1(x_1) = 0$ y $s''_{N-1}(x_N) = 0$. Estas condiciones dan un sistema completo de $4N - 4$ ecuaciones con $4N - 4$ incógnitas, que se puede resolver utilizando la eliminación de Gauss para calcular todos los coeficientes que describe la spline.

Sin embargo es posible calcular la misma spline de forma más eficaz, dado que realmente sólo hay $N - 2$ «incógnitas»: la mayoría de las condiciones de la spline son redundantes. Por ejemplo, suponiendo que p_i es el valor de la derivada segunda de la spline en x_i , se tiene que $s''_{i-1}(x_i) = s''_i(x_i) = p_i$ para $i = 2, \dots, N - 1$, con $p_1 = p_N = 0$. Si los valores de p_1, \dots, p_N son conocidos, entonces todos los coeficientes a, b, c, d se pueden calcular por segmentos de la spline, dado que se tienen cuatro ecuaciones con cuatro incógnitas para cada segmento de spline: para $i = 1, 2, \dots, N - 1$, se debe tener

$$s_i(x_i) = y_i$$

$$s_i(x_{i+1}) = y_{i+1}$$

$$s''_i(x_i) = p_i$$

$$s''_i(x_{i+1}) = p_{i+1}$$

Como los valores de x e y son dados, para determinar completamente la spline se necesita solamente calcular los valores de p_2, \dots, p_{N-1} . Para hacer esto, se utiliza la condición de que las derivadas primeras deben coincidir: estas $N - 2$ condiciones proporcionan exactamente las $N - 2$ ecuaciones que se necesitan para resolver las $N - 2$ incógnitas, los valores de las p_i derivadas segundas.

Para expresar los coeficientes a, b, c y d en términos de los valores de las p derivadas segundas, se pueden sustituir estas expresiones en las cuatro ecuaciones anteriores para cada segmento de spline, obteniéndose algunas expresiones innecesariamente complicadas. En su lugar, es conveniente expresar las ecuaciones para los segmentos de spline en una forma canónica que implica una menor cantidad de coeficientes desconocidos. Si se hace un cambio de variable $t = (x - x_i)/(x_{i+1} - x_i)$ entonces la spline se puede expresar como:

$$s_i(t) = ty_{i+1} + (1 - t)y_i + (x_{i+1} - x_i)^2 ((t^3 - t)p_{i+1} - ((1 - t)^3 - (1 - t))p_i)/6.$$

Ahora cada spline está definida en el intervalo $[0, 1]$. Esta ecuación es menos complicada de lo que parece, porque lo único que interesa son los puntos extremos 0 y 1 y tanto t como $(1 - t)$ son 0 en estos puntos. Esta representación permite comprobar fácilmente que la spline interpola correctamente y es continua, porque $s_{i-1}(1) = s_i(0) = y_i$ para $i = 2, \dots, N - 1$, y esto es sólo un tanto más complicado que comprobar que las derivadas segundas son continuas porque $s'_i(1) = s''_{i+1}(0) = p_{i+1}$. Éstos son polinomios (de tercer grado) cúbicos que satisfacen las condiciones de los extremos, por lo que son equivalentes a los segmentos de

spline descritos anteriormente. Si se substituyeran por t y se encontraran los coeficientes de x^3 , etc., se obtendrían las mismas expresiones de a , b , c y d en términos de x , y y p como si se hubiera utilizado el método descrito en el párrafo anterior. Pero no existe ninguna razón para hacer esto, dado que se ha comprobado que estos segmentos de spline satisfacen las condiciones de los extremos y que se puede evaluar cada uno en todo punto de su intervalo calculando t y utilizando la fórmula anterior (una vez conocidos los valores de p).

Para determinar las p se necesita imponer que las primeras derivadas de los segmentos de spline sean iguales en los extremos. La primera derivada (con respecto a x) de la ecuación anterior es

$$s_i'(t) = z_i + (x_{i+1} - x_i)((3t^2 - 1)p_{i+1} + (3(1-t)^2 - 1)p_i)/6$$

donde $z_i = (y_{i+1} - y_i)/(x_{i+1} - x_i)$. Ahora, haciendo $s'_{i-1}(1) = s'_i(0)$ para $i = 2, \dots, N-1$ se obtiene el sistema de $N-2$ ecuaciones deseado:

$$(x_i - x_{i-1})p_{i-1} + 2(x_{i+1} - x_{i-1})p_i + (x_{i+1} - x_i)p_{i+1} = 6(z_i - z_{i-1}).$$

Este sistema de ecuaciones es una sencilla forma tridiagonal que se resuelve fácilmente por medio de una versión simplificada de la eliminación de Gauss, como se vio en el Capítulo 37. Si se pone $u_i = x_{i+1}$, $d_i = 2(x_{i+1} - x_{i-1})$ y $w_i = 6(z_i - z_{i-1})$, se tiene, por ejemplo, el sistema de ecuaciones simultáneas siguiente para $N = 7$.

$$\begin{pmatrix} d_2 & u_2 & 0 & 0 & 0 \\ u_2 & d_3 & u_3 & 0 & 0 \\ 0 & u_3 & d_4 & u_4 & 0 \\ 0 & 0 & u_4 & d_5 & u_5 \\ 0 & 0 & 0 & u_5 & d_6 \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{pmatrix} = \begin{pmatrix} w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{pmatrix}.$$

De hecho, este es un sistema tridiagonal simétrico, en el que las diagonales que están debajo de la principal son iguales a las que están por encima. Se deduce que no es necesario pivotar alrededor del mayor de los elementos que existen para obtener una solución exacta de este sistema de ecuaciones.

El método descrito en el párrafo anterior para calcular una spline cúbica se traduce muy fácilmente en C++:

```
spline(float x[], float y[], int N)
{
    for (i = 2; i < N; i++) d[i] = 2*(x[i+1]-x[i-1]);
    for (i = 1; i < N; i++) u[i] = x[i+1]-x[i];
    for (i = 2; i < N; i++)
        w[i] = 6.0*((y[i+1]-y[i])/u[i])
```

```

        -(y[i]-y[i-1])u[i-1]);
p[1] = 0.0; p[N] = 0.0;
for (i = 2; i < N-1; i++)
{
    w[i+1] = w[i+1] - w[i]*u[i]/d[i];
    d[i+1] = d[i+1] - u[i]*u[i]/d[i];
}
for (i = N-1; i > 1; i--)
    p[i] = (w[i]-u[i]*p[i+1])/d[i];
}

```

Los arrays d y u son la representación de la matriz tridiagonal que se resuelve utilizando el programa del Capítulo 37. Aquí se reemplaza $d[i]$ por $a[i][i]$, $u[i]$ por $a[i+1][i]$ o $a[i][i+1]$ y $z[i]$ por $a[i][N+1]$.

Propiedad 38.1 *Una spline cúbica de N puntos puede construirse en tiempo lineal.*

Esta propiedad es evidente según el propio programa, que es simplemente una sucesión de recorridos lineales a través de los datos.■

Como ejemplo de la construcción de una spline cúbica se considera el ajuste de una spline de cinco puntos

$$(1.0, 2.0), (2.0, 1.5), (4.0, 1.25), (5.0, 1.2), (8.0, 1.125), (10.0, 1.1)$$

(Estos puntos provienen de la función $1 + 1/x$.) Los parámetros de la spline se encuentran resolviendo el sistema de ecuaciones

$$\begin{pmatrix} 6 & 2 & 0 & 0 \\ 2 & 6 & 1 & 0 \\ 0 & 1 & 8 & 3 \\ 0 & 0 & 3 & 10 \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \\ p_4 \\ p_5 \end{pmatrix} = \begin{pmatrix} 2.250 \\ 0.450 \\ 0.150 \\ 0.075 \end{pmatrix}$$

lo que da el resultado $p_2 = 0.39541$, $p_3 = -0.06123$, $p_4 = 0.02658$, $p_5 = -0.00047$.

Para evaluar la spline para cualquier valor de x del intervalo $[x_1, x_N]$, basta simplemente con encontrar el intervalo $[x_i, x_{i+1}]$ que contiene a x , y después calcular t y utilizar la fórmula anterior para $s_i(x)$ (que a su vez utiliza los valores calculados de p_i y p_{i+1}).

```

float f(float x)
{ return x*x*x - x; }
float eval (float v)

```

```

{
    float t; int i = 1;
    while (v > x[i+1]) i++;
    t = (v-x[i])/u[i];
    return t*y[i+1] + (1-t)*y[i] +
        u[i]*u[i]*(f(t)*p[i+1]+f(1-t)*p[i])/6.0;
}

```

Este programa no verifica la condición de error cuando v no está entre $x[1]$ y $x[N]$. Si el número de segmentos de la spline es grande (esto es, si N es grande), se podría utilizar alguno de los métodos de búsqueda más eficaces del Capítulo 14 para encontrar el intervalo que contenga a v .

Existen numerosas variantes de la idea del ajuste de curvas por unión de polinomios de forma «continua»: el cálculo de splines es un campo de estudio muy desarrollado. Cada tipo de spline implica distintos criterios de continuidad, así como cambios tales como la irritante condición de que cada spline debe tocar exactamente a cada punto de los datos. Desde el punto de vista del cálculo, las splines implican exactamente los mismos pasos para determinar los coeficientes de cada segmento de spline por medio de la resolución del sistema de ecuaciones lineales que se deriva de imponer restricciones al modo en el que se han unido.

Método de los mínimos cuadrados

Frecuentemente sucede que, incluso aunque los datos no sean del todo exactos, se tiene alguna idea de la forma de la función que se va a ajustar. La función puede depender de ciertos parámetros

$$f(x) = f(c_1, c_2, \dots, c_M, x)$$

y el procedimiento de ajuste de curvas debe encontrar los valores de los parámetros que hagan la «mejor» concordancia entre los valores observados y los puntos dados. Si la función fuera un polinomio (en cuyo caso los parámetros serían sus coeficientes) y sus valores fueran exactos, entonces esto sería una interpolación. Pero ahora se están considerando funciones más generales y datos inexactos. Para simplificar la presentación, este estudio se va a limitar al ajuste de funciones expresadas como combinaciones lineales de funciones simples, cuyos coeficientes son los parámetros desconocidos:

$$f(x) = c_1 f_1(x) + c_2 f_2(x) + \dots + c_M f_M(x)$$

Esto engloba a la mayor parte de las funciones en las que se está interesado. Una vez estudiado este caso, se tratarán funciones más generales.

Una forma común de medir la precisión del ajuste es el *criterio de los mínimos cuadrados*. Aquí el error se calcula sumando los cuadrados de los errores de cada punto de observación:

$$E = \sum_{1 \leq j \leq N} (f(x_j) - y_j)^2.$$

Esta medida es bastante natural: se eleva al cuadrado para eliminar las anulaciones entre errores de signo diferente. Evidentemente es más deseable determinar los valores de los parámetros que minimizar E . Por ello esta elección se puede efectuar con eficiencia por medio del denominado *método de los mínimos cuadrados*.

Este método sigue bastante fielmente la definición. Para simplificar las explicaciones se considera el caso $M = 2$, $N = 3$, pero el método general se puede deducir directamente. Se supone que se tienen tres puntos x_1 , x_2 , x_3 y los valores correspondientes y_1 , y_2 , y_3 que se deben ajustar a una función de la forma $f(x) = c_1 f_1(x) + c_2 f_2(x)$. El objetivo es encontrar los valores de los coeficientes c_1 y c_2 que minimicen el error de los mínimos cuadrados

$$\begin{aligned} E &= (c_1 f_1(x_1) + c_2 f_2(x_1) - y_1)^2 \\ &\quad + (c_1 f_1(x_2) + c_2 f_2(x_2) - y_2)^2 \\ &\quad + (c_1 f_1(x_3) + c_2 f_2(x_3) - y_3)^2. \end{aligned}$$

Para encontrar los valores de c_1 y c_2 que minimizan este error, es suficiente con imponer que las derivadas dE/dc_1 y dE/dc_2 sean nulas. Para c_1 se tiene:

$$\begin{aligned} dE/dc_1 &= 2(c_1 f_1(x_1) + c_2 f_2(x_1) - y_1) f_1'(x_1) \\ &\quad + 2(c_1 f_1(x_2) + c_2 f_2(x_2) - y_2) f_1'(x_2) \\ &\quad + 2(c_1 f_1(x_3) + c_2 f_2(x_3) - y_3) f_1'(x_3). \end{aligned}$$

Al hacer igual a cero las derivadas, las variables c_1 y c_2 deben satisfacer la siguiente ecuación (en la que las $f_i(x_i)$ son todas «constantes» con valores conocidos):

$$\begin{aligned} c_1(f_1(x_1)f_1'(x_1) + f_1(x_2)f_1'(x_2) + f_1(x_3)) \\ + c_2(f_2(x_1)f_1'(x_1) + f_2(x_2)f_1'(x_2) + f_2(x_3)f_1'(x_3)) \\ = y_1 f_1'(x_1) + y_2 f_1'(x_2) + y_3 f_1'(x_3). \end{aligned}$$

Se obtiene una ecuación similar cuando se anula la derivada dE/dc_2 . Estas ecua-

ciones tan impresionantes se pueden simplificar en gran medida utilizando la notación vectorial y la operación «producto escalar». Si se definen los vectores $\mathbf{x} = (x_1, x_2, x_3)$ y $\mathbf{y} = (y_1, y_2, y_3)$, entonces el producto escalar de \mathbf{x} e \mathbf{y} será el número real definido por

$$\mathbf{x} \cdot \mathbf{y} = x_1y_1 + x_2y_2 + x_3y_3$$

Si ahora se definen los vectores $\mathbf{f}_1 = (f_1(x_1), f_1(x_2), f_1(x_3))$ y $\mathbf{f}_2 = (f_2(x_1), f_2(x_2), f_2(x_3))$, entonces las ecuaciones para los coeficientes c_1 y c_2 se pueden expresar de forma muy simple:

$$\begin{aligned} c_1\mathbf{f}_1 \cdot \mathbf{f}_1 + c_2\mathbf{f}_1 \cdot \mathbf{f}_2 &= \mathbf{y} \cdot \mathbf{f}_1, \\ c_1\mathbf{f}_2 \cdot \mathbf{f}_1 + c_2\mathbf{f}_2 \cdot \mathbf{f}_2 &= \mathbf{y} \cdot \mathbf{f}_2. \end{aligned}$$

Éstas se pueden resolver con la eliminación de Gauss para encontrar los coeficientes deseados.

Por ejemplo, supóngase que se sabe que los puntos

$$(1.0, 2.05) (2.0, 1.53) (4.0, 1.26) (5.0, 1.21) (8.0, 1.13) (10.0, 1.1)$$

se deben ajustar por una función de la forma $c_1 + c_2/x$. (Estos datos son ligeras perturbaciones de los valores exactos de $1 + 1/x$.) En este caso, f_1 es una constante ($\mathbf{f}_1 = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0)$) y $\mathbf{f}_2 = (1.0, 0.5, 0.25, 0.2, 0.125, 0.1)$, por lo que hay que resolver el sistema de ecuaciones

$$\begin{pmatrix} 6.000 & 2.175 \\ 2.175 & 1.378 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 8.280 \\ 3.623 \end{pmatrix}$$

lo que da el resultado $c_1 = 0.998$ y $c_2 = 1.054$ (los dos valores están próximos a la unidad, como era de esperarse).

El método que se acaba de esbozar se generaliza fácilmente para encontrar más de dos coeficientes. Para encontrar las constantes c_1, c_2, \dots, c_M de

$$f(x) = c_1 f_1(x) + c_2 f_2(x) + \dots + c_M f_M(x)$$

que minimizan los errores de los mínimos cuadrados para los vectores de puntos y de observación

$$\begin{aligned} \mathbf{x} &= (x_1, x_2, \dots, x_N), \\ \mathbf{y} &= (y_1, y_2, \dots, y_N), \end{aligned}$$

se comienza por calcular los vectores componentes de las funciones

$$\begin{aligned}\mathbf{f}_1 &= (f_1(x_1), f_1(x_2), \dots, f_1(x_N)), \\ \mathbf{f}_2 &= (f_2(x_1), f_2(x_2), \dots, f_2(x_N)), \\ &\vdots \\ &\vdots \\ \mathbf{f}_m &= (f_m(x_1), f_m(x_2), \dots, f_m(x_N)).\end{aligned}$$

A continuación se construye un sistema de $M * M$ ecuaciones lineales $Ac = b$ con

$$a_{ij} = \mathbf{f}_i \cdot \mathbf{f}_j,$$

$$b_j = \mathbf{f}_j \cdot \mathbf{y}.$$

La solución de este sistema de ecuaciones simultáneas proporciona los coeficientes buscados.

Este método es fácil de implementar manteniendo un array de dos dimensiones para los vectores \mathbf{f} , considerando \mathbf{a} y como el $(M + 1)$ -ésimo vector. Dicho array se puede construir, de acuerdo con la descripción anterior, por medio del siguiente código:

```
for (i = 1; i <= M; i++)
    for (j = 1; j <= M+1; j++)
    {
        t = 0.0;
        for (k = 1; k <= N; k++)
            t += f[i][k]*f[j][k];
        a[i][j] = t;
    }
```

y el correspondiente sistema de ecuaciones simultáneas se puede resolver utilizando el procedimiento de eliminación de Gauss del Capítulo 37.

El método de los mínimos cuadrados se puede extender a funciones no lineales (como, por ejemplo, la función $f(x) = c_1 e^{-c_2 x} \operatorname{senc}_3 x$), y se utiliza a menudo para este tipo de aplicación. La idea es fundamentalmente la misma; el problema es que las derivadas no son siempre fáciles de calcular. En estos casos se utiliza un método *iterativo*: partir de estimaciones de los coeficientes y después utilizarlas en el método de los mínimos cuadrados para calcular las derivadas y obtener así una mejor estimación de los coeficientes. Este método básico, muy utilizado hoy, fue esbozado por Gauss en la década de 1820.

Ejercicios

1. Aproximar la función $\lg x$ con un polinomio de interpolación de cuarto grado en los puntos 1, 2, 3, 4 y 5. Estimar la calidad del ajuste calculando la suma de los cuadrados de los errores en 1.5, 2.5, 3.5 y 4.5.
2. Resolver el ejercicio anterior para la función $\sin x$. Dibujar la función y su aproximación (si es posible) en la computadora.
3. Resolver los ejercicios anteriores utilizando una spline cúbica en lugar de interpolaciones polinómicas.
4. Aproximar la función $\lg x$ con una spline cúbica con nudos en 2^N para N entre 1 y 10. Ensayar con diferentes colocaciones de los nudos en el mismo intervalo para intentar obtener un ajuste mejor.
5. ¿Qué sucederá en el método de ajuste de datos por mínimos cuadrados si una de las funciones es $f_i(x) = 0$, para un i dado?
6. Utilizar el método de ajuste de curvas por mínimos cuadrados para encontrar los valores de a y b que proporcionen la mejor fórmula del tipo $aN \ln N + bN$ para describir el número total de instrucciones ejecutadas al aplicar el método de Quicksort a un archivo aleatorio.
7. ¿Qué valores de a , b , c minimizan el error de los mínimos cuadrados cuando se utiliza la función $f(x) = ax \log x + bx + c$ para aproximar las observaciones $f(1) = 0$, $f(4) = 13$, $f(8) = 41$?
8. Excluyendo la fase de eliminación de Gauss, ¿cuántas multiplicaciones son necesarias para encontrar M coeficientes a partir de N observaciones por el método de los mínimos cuadrados?
9. ¿En qué circunstancias podría ser singular la matriz generada en el método de ajuste de curvas de los mínimos cuadrados?
10. ¿Puede funcionar el método de los mínimos cuadrados cuando se incluyen dos observaciones diferentes para el mismo punto?

Integración

El cálculo de una integral es una de las operaciones analíticas fundamentales que se lleva a cabo frecuentemente en el procesamiento de funciones por medio de computadoras. Lo que se desea es encontrar el «área situada bajo la curva» con eficacia y un grado razonable de precisión. En este capítulo se examinarán ciertos algoritmos clásicos de resolución de este problema numérico elemental.

En primer lugar se hará una breve presentación del caso en el que se dispone de una representación explícita de la función. En estos casos puede ser posible hacer una *integración simbólica* para transformar la representación de la función en una representación similar de la integral. Esto es posible cuando las funciones a procesar forman parte de la clase restringida de funciones, cuyas integrales están definidas analíticamente o en el contexto de sistemas que procesan tales representaciones de funciones.

En el extremo opuesto, se pueden definir las funciones por medio de una tabla, de forma que sus valores se conocen solamente para un pequeño número de puntos. En tales casos sólo se puede dar un valor aproximado de la integral, basándose en hipótesis sobre el comportamiento de la función entre los puntos. La precisión de la integral depende casi exclusivamente de la validez de las hipótesis.

La situación más frecuente se encuentra entre estos casos extremos: la función a integrar se representa de tal manera que se puede calcular su valor en cualquier punto. Aquí, otra vez, la precisión de la integral depende de las hipótesis sobre el comportamiento de la función entre los puntos cualesquiera que se elijan para la evaluación. El objetivo consiste en calcular una aproximación razonable de la integral de la función, sin llevar a cabo un excesivo número de evaluaciones de la función. En análisis numérico, a este cálculo a menudo se le denomina *cuadratura*.

En este capítulo se examinarán varios métodos de cuadratura elementales —el objetivo es adquirir alguna experiencia con los métodos fundamentales de cálculo numérico—. Muchas aplicaciones pueden realmente beneficiarse de la correcta aplicación de las técnicas elementales que se van a considerar, pero los

métodos de resolución de problemas más avanzados, en especial la solución numérica de ecuaciones diferenciales, son los más importantes en la práctica.

Integración simbólica

Si se dispone de una completa información respecto a una función se puede obtener provecho del empleo de un método que implique la manipulación de alguna representación de la función, en lugar de trabajar con valores numéricos. El objetivo consiste en transformar una representación de la función en una representación de la integral, exactamente de la misma forma que cuando se calcula manualmente una integral indefinida.

La integración de polinomios es un simple ejemplo de esta técnica. En el Capítulo 36 se examinaron métodos de cálculo «simbólico» de sumas y productos de polinomios, utilizando programas que funcionan con una representación particular de los polinomios y generando la representación de las respuestas a partir de la representación de los datos de entrada. La operación de integración (y de diferenciación) de polinomios se puede realizar también de esta forma. Si un polinomio

$$p(x) = p_0 + p_1x + p_2x^2 + \dots + p_{N-1}x^{N-1}$$

está representado simplemente manteniendo los valores de sus coeficientes en un array p entonces se puede calcular fácilmente su integral de la siguiente forma:

```
for (i = N; i > 0; i--) p[i] = p[i-1]/i; p[0] = 0;
```

Para cada término del polinomio este programa aplica la muy conocida regla de integración simbólica $\int_0^x t^{i-1} dt = x^i/i$ para $i > 0$. Se puede integrar una clase de funciones más amplia que los simples polinomios añadiendo más reglas simbólicas. La adición de reglas compuestas tales como la *integración por partes*,

$$\int u \, dv = uv - \int v \, du,$$

puede ampliar de gran forma el número de funciones que se pueden tratar. (La integración por partes requiere la capacidad de derivar. La derivación simbólica es claramente más fácil que la integración simbólica, porque un conjunto razonable de reglas elementales, más la *regla de derivación en cadena* de funciones compuestas, es suficiente para la mayor parte de las funciones comunes.)

El gran número de reglas que se pueden aplicar a una función particular hace de la integración simbólica una tarea difícil. Incluso hasta hace poco tiempo no se había podido probar la existencia de un *algoritmo* para esta tarea: un proce-

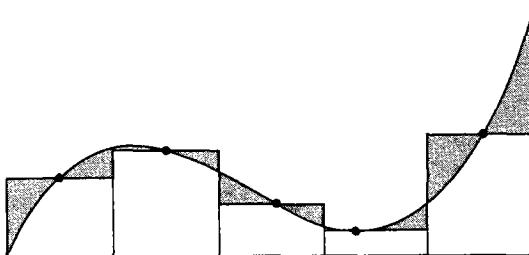


Figura 39.1 Regla del rectángulo.

dimiento que o bien devuelve la integral de cualquier función dada, o indica si la respuesta no se puede expresar en términos de funciones elementales. Una descripción de este algoritmo en su totalidad está fuera del alcance de este libro. Sin embargo, cuando las funciones a procesar pertenecen a una clase restringida, la integración simbólica puede ser una herramienta potente.

Por supuesto, las técnicas simbólicas tienen el inconveniente fundamental de que un gran número de integrales (muchas de las cuales aparecen en la práctica) no se pueden evaluar simbólicamente. A continuación se examinarán algunas técnicas que se han desarrollado para calcular aproximaciones de los valores de integrales reales.

Métodos de cuadratura elementales

La forma más evidente de aproximar el valor de una integral es posiblemente el *método del rectángulo*. La evaluación de una integral es lo mismo que calcular el área de la parte inferior de la curva, y se puede estimar esta cantidad sumando las áreas de los pequeños rectángulos subtendidos por la curva, como los presentados en la Figura 39.1.

Con mayor precisión, supóngase que se calcula $\int_a^b f(x)dx$, y que se divide el intervalo de integración $[a,b]$ en N partes, delimitadas por los puntos x_1, x_2, \dots, x_{N+1} . Así se obtienen N rectángulos, con un ancho del rectángulo i -ésimo, ($1 \leq i \leq N$) dado por $x_{i+1} - x_i$. Para la altura del rectángulo i -ésimo, se podría utilizar $f(x_i)$ o $f(x_{i+1})$, pero parece que el resultado es más preciso si se utiliza el valor de f de la mitad del intervalo ($f((x_i + x_{i+1})/2)$), como en el diagrama anterior. Esto conduce a la fórmula de cuadratura

$$r = \sum_{1 \leq i \leq N} (x_{i+1} - x_i) f\left(\frac{x_i + x_{i+1}}{2}\right)$$

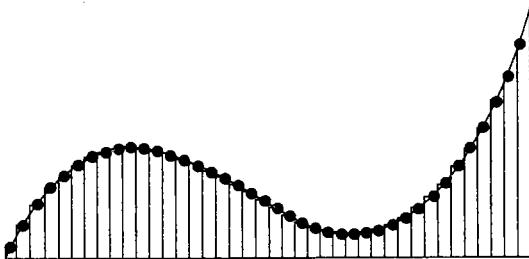


Figura 39.2 Regla del rectángulo con menores tamaños de los intervalos.

que estima el valor de la integral de $f(x)$ sobre el intervalo entre $a = x_1$ y $b = x_{N+1}$. En el caso más común, en el que los intervalos son del mismo tamaño, es decir $x_{i+1} - x_i = w$, se tiene $x_{i+1} + x_i = (2i + 1)w$, por lo que se puede calcular fácilmente la aproximación r de la integral.

```
double intrect(double a, double b, int N)
{
    int i; double r = 0; double w = (b-a)/N;
    for (i = 1; i <= N; i++) r += w*f(a-w/2+i*w);
    return r;
}
```

Por supuesto, a medida que crece N , la respuesta se hace más exacta. La Figura 39.2 muestra el resultado de utilizar intervalos de pequeño tamaño para la función representada en la Figura 39.1.

A continuación se da un ejemplo más cuantitativo, que muestra la estimación producida por esta función para $\int_1^2 dx/x$ (que se sabe que es igual a $\ln 2 = 0.6931471805599\dots$) cuando se invoca con la llamada a `intrect(1.0, 2.0, N)` para $N = 10, 100, 1000$:

10	0.6928353604100
100	0.6931440556283
1000	0.6931471493100

Cuando $N = 1000$, la respuesta es exacta hasta aproximadamente el séptimo decimal. Los métodos de cuadratura más sofisticados pueden proporcionar más exactitud con menos trabajo.

El análisis de la estimación de error de cada método particular puede sugerir a menudo técnicas más precisas. La expresión analítica del error producido por el método del rectángulo se obtiene desarrollando $f(x)$ en serie de Taylor en la

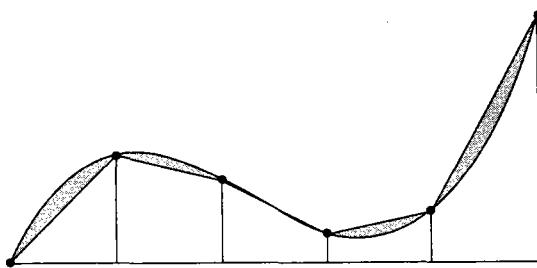


Figura 39.3 Regla del trapecio.

proximidad del punto medio de cada intervalo, integrando y sumando sobre todos los intervalos. Si entrar en detalles de los cálculos se obtiene que

$$\int_a^b f(x)dx = r + w^3 e_3 + w^5 e_5 + \dots$$

donde w es la anchura del intervalo ($(b - a)/N$) y e_3 depende del valor de la tercera derivada de f en los puntos medios del intervalo, etc. (Ésta es normalmente una buena aproximación, porque la mayor parte de las funciones «razonables» tienen derivadas de orden superior de pequeño valor absoluto, a pesar de que esto no siempre es cierto.) Por ejemplo, si se elige $w = 0.01$ (lo que corresponde a $N = 200$ en el ejemplo anterior), esta fórmula dice que la integral calculada por el procedimiento anterior debe ser exacta hasta aproximadamente el sexto decimal.

Otra forma de aproximar la integral consiste en dividir el área bajo la curva en trapecios, como se muestra en la Figura 39.3. Recuérdese que el área del trapecio es igual a la mitad del producto de la altura por la suma de las longitudes de las dos bases. El *método del trapecio* conduce a la fórmula de cuadratura

$$t = \sum_{1 \leq i \leq N} (x_{i+1} - x_i) \frac{f(x_i) + f(x_{i+1})}{2}.$$

El siguiente procedimiento implementa el método del trapecio en el caso tipo en el que todos los intervalos tienen la misma anchura:

```
double inttrap (double a, double, b, int N)
{
    int i; double t = 0; double w = (b-a)/N;
    for (i = 1; i <=N; i++)
        t += w * (f(a+i*w) + f(a+(i+1)*w))/2;
}
```

```

    t += w*(f(a+(i-1)*w)+f(a+i*w))/2;
    return t;
}

```

El error de este método se puede obtener de forma similar al del método del rectángulo. Se encuentra que

$$\int_a^b f(x)dx = t - 2w^3 e_3 - 4w^5 e_5 + \dots$$

Así el método del rectángulo es dos veces más preciso que el del trapecio. Esto se confirma por el ejemplo: este procedimiento genera las estimaciones siguientes para $\int_1^2 dx/x$:

10	0,6937714031754
100	0,6931534304818
1000	0,6931472430599

En un principio puede parecer sorprendente que el método del rectángulo sea más preciso que el del trapecio. Sin embargo, recuérdese que los rectángulos tienden a estar parte sobre la curva y parte bajo ella (por lo que el error del interior del intervalo se puede compensar con el del exterior), mientras que los trapecios tienden a estar o completamente sobre la curva o completamente bajo ella. La Figura 39.4 muestra el método del trapecio con un pequeño tamaño de intervalo —parece ajustar la curva exactamente, pero de hecho la Figura 39.2 proporciona una mejor estimación del área situada bajo la curva.

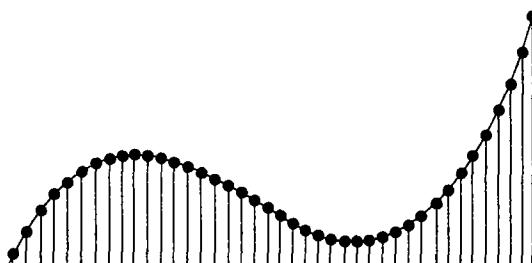


Figura 39.4 Regla del trapecio con menores tamaños de los intervalos.

Otro método bastante aceptable es el de la *cuadratura por spline*: se lleva a cabo una interpolación spline utilizando los métodos presentados en el capítulo anterior, y después se calcula la integral aplicando, segmento a segmento, la sencilla técnica de integración polinómica simbólica antes descrita. Este mé-

todo es realmente muy parecido a los de las reglas del rectángulo y del trapecio, como se verá a continuación.

Métodos compuestos

El examen de las fórmulas de error de los métodos del rectángulo y trapecio presentados anteriormente conduce a un método sencillo, denominado *método de Simpson*, que proporciona una considerable precisión. La idea es eliminar el término dominante del error por medio de una combinación de los métodos citados. Multiplicando por dos la fórmula del método del rectángulo, sumando la del trapecio y dividiendo después por tres se obtiene la ecuación

$$\int_a^b f(x)dx = \frac{1}{3} (2r + t - 2w^5 e_5 + \dots).$$

El término w^3 ha desaparecido, por lo que esta fórmula indica que se puede obtener un método con una precisión del orden de w^5 combinando las fórmulas de cuadratura de la misma forma:

$$s = \sum_{1 \leq i \leq N} \frac{x_{i+1} - x_i}{6} \left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1}) \right).$$

Si se utiliza un intervalo de tamaño 0,01 con el método de Simpson, se puede calcular la integral con una precisión del orden de 10 dígitos. Aquí, otra vez, esto se demuestra en el ejemplo. La implementación del método de Simpson es solamente algo más complicada que las otras (como antes, se consideran intervalos que tienen todos la misma anchura):

```
double intsimp (double a, double b, int N)
{
    int i; double s = 0; double w = (b-a)/N;
    for (i = 1; i <= N; i++)
        s += w*(f(a+(i-1)*w) +
               4*f(a-w/2+i*w))
               f(a+i*w)/6;
    return s;
}
```

Este programa necesita tres «evaluaciones de función» (en lugar de dos) en el bucle interno, pero proporciona resultados más precisos que los dos métodos anteriores:

10	0,6931473746651
100	0,6931471805795
1000	0,6931471805599

Se han desarrollado otros métodos de cuadratura más complicados, que ganan precisión al combinar métodos simples con errores similares. El más conocido es la *integración de Romberg*, que utiliza dos conjuntos diferentes de intervalos de subdivisión para sus dos «métodos».

Se deduce que el método de Simpson es exactamente equivalente a interpolar los datos por una función cuadrática por partes y después integrar. Es interesante destacar que los cuatro métodos que se han presentado se pueden describir todos como métodos de interpolación por partes: la regla del rectángulo interpola por una constante (polinomios de grado cero); la del trapecio por una recta (polinomio de grado uno); la de Simpson por un polinomio de segundo grado, y el método de la spline por un polinomio cúbico.

Cuadratura adaptativa

Uno de los mayores defectos de los métodos que se acaban de presentar es que los errores que los acompañan dependen no solamente del tamaño de los subintervalos utilizados, sino también del valor de las derivadas de orden superior de la función integrada. Esto significa que estos métodos no serán válidos para determinadas funciones (las que tengan valores grandes en las derivadas de orden superior). Pero son muy pocas las funciones que tienen grandes valores en la mayor parte de las derivadas de orden superior. Es razonable utilizar pequeños intervalos allí donde las derivadas son grandes, y grandes intervalos allí donde son pequeñas. Uno de los métodos que sigue sistemáticamente esta regla es el procedimiento denominado *cuadratura adaptativa*.

El principio general de la cuadratura adaptativa es utilizar dos métodos de cuadratura diferentes para cada intervalo de subdivisión, comparar los resultados y subdividir otra vez el intervalo si la diferencia es demasiado grande. Por supuesto, es preciso tener mucho cuidado al hacerlo, dado que, si se utilizan dos métodos «malos» similares, pueden «ponerse de acuerdo» para dar un mal resultado. Una forma de evitar esto consiste en asegurarse de que un método sobreestima siempre los resultados, y que el otro siempre los subestima. Otra forma de evitarlo es asegurarse de que uno de los métodos es mucho más preciso que el otro. Es precisamente una metodología de este último tipo la que se va a presentar a continuación.

La subdivisión recursiva del intervalo se acompaña de importantes costes de operatividad, por lo que se debe utilizar un buen método para la estimación de las integrales, como en la implementación siguiente:

```
double adapt (double a, double b)
```

```

{
    double x = intsimp(a, b, 10);
    if (fabs(x - intsimp (a, b, 5)) > tolerancia)
        return adapt (a, (a+b)/2) + adapt ((a+b)/2, b);
    return x;
}

```

Las dos estimaciones de la integral se obtienen por el método de Simpson, pero una de ellas utiliza dos veces más subintervalos que la otra. En esencia, esta técnica comprueba la precisión del método de Simpson en cada intervalo en cuestión, y luego vuelve a subdividir, si no es lo bastante preciso.

A diferencia de los métodos anteriores en los que primero se decidía la cantidad de trabajo a hacer y luego se recogían los resultados, independientemente de la precisión, el método de la cuadratura adaptativa hace todo el trabajo que sea necesario para alcanzar un grado de precisión determinado con anterioridad. Esto significa que se debe elegir el valor de tolerancia con mucho cuidado, para que la función no entre en un bucle indefinido al intentar alcanzar una precisión elevadísima. El número de pasos que se necesitan depende mucho de la naturaleza de la función a integrar. Una función que fluctúe ampliamente necesita un gran número de pasos, pero una tal función provocará también resultados muy imprecisos con los métodos de «intervalos fijos». La Figura 39.5 muestra los puntos evaluados cuando se utiliza la cuadratura adaptativa (basada en la regla del trapezio) en la función de las Figuras 39.1 a 39.4. Obsérvese que los intervalos son mayores donde la función es directa y continua, y más pequeños donde la pendiente es más fuerte.

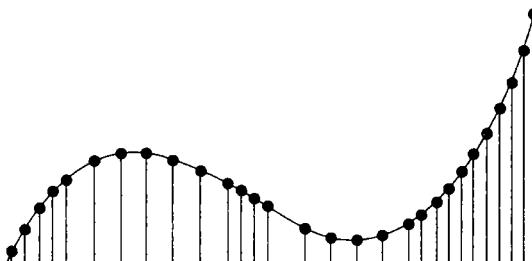


Figura 39.5 Cuadratura adaptativa.

Una buena función como la del ejemplo puede manipularse con un número razonable de pasos. La tabla siguiente proporciona, para diferentes valores de t el valor generado y el número de llamadas recursivas que se necesitan para calcular $\int_1^2 dx/x$:

0,00000010000	0,6931471829695	5
0,00000000100	0,6931471806413	13
0,00000000001	0,6931471805623	33

El programa anterior se puede mejorar de varias formas. Primero, los valores de la función para la llamada a `intsimp(a, b, 10)` pueden dividirse en `intsimp(a,b,5)`.

Segundo, los límites de tolerancia pueden estar más relacionados con la precisión de la respuesta si `tolerancia` se escala con respecto a la relación entre el tamaño del intervalo actual y el del intervalo completo. También se puede mejorar de forma evidente la rutina utilizando una regla de cuadratura en vez de la de Simpson (pero una ley básica de la recursión dice que no es una buena idea utilizar cualquier otra técnica de *cuadratura adaptativa*). Una rutina más sofisticada de la cuadratura adaptativa puede proporcionar resultados más exactos para problemas que no se pueden tratar de otra manera, pero debe prestarse mucha atención a los tipos de función que se han de procesar.

El paradigma de diseño de algoritmos de «divide y vencerás» es también útil en los programas numéricos. De hecho, los métodos adaptativos de este tipo son muy importantes como soluciones técnicas para problemas numéricos tales como la integración en grandes dimensiones y la resolución numérica de ecuaciones diferenciales.

Ejercicios

1. Escribir un programa para integrar (y derivar) simbólicamente polinomios en x y $\ln x$. Utilizar una implementación recursiva basada en la integración por partes.
2. ¿Qué método de cuadratura es susceptible de dar el mejor resultado al integrar las funciones siguientes: $f(x) = 5x$, $f(x) = (3 - x)(4 + x)$, $f(x) = \sin(x)$?
3. Obtener el resultado de utilizar cada uno de los cuatro métodos de cuadratura (rectángulo, trapecio, Simpson y spline) para integrar $y = 1/x$ en el intervalo $[0.1, 10]$.
4. La misma pregunta anterior para la función $y = \sin x$.
5. Explicar lo que sucede cuando se utiliza la cuadratura adaptativa para integrar la función $y = 1/x$ en el intervalo $[-1, 2]$.
6. Responder a la pregunta anterior para los métodos de cuadratura elementales.
7. Obtener los puntos de evaluación al utilizar la cuadratura adaptativa para integrar la función $y = 1/x$ en el intervalo $[0.1, 10]$ con una tolerancia de 0.1.
8. Comparar la precisión de una cuadratura adaptativa basada en el método de Simpson con otra basada en el del rectángulo para la integral dada en el ejercicio anterior.

9. Responder a la pregunta anterior para la función $y = \operatorname{sen}x$.
10. Dar un ejemplo concreto de una función para la que la cuadratura adaptativa es susceptible de dar un resultado drásticamente más preciso que los otros métodos.

REFERENCIAS para Algoritmos matemáticos

Gran parte de los temas de esta sección están dentro del dominio del análisis numérico, para el que existen varias obras excelentes, como por ejemplo el libro de Conte y de Boor. Una obra que dedica una particular atención a los problemas de cálculo numérico es la de Forsythe, Malcomb y Moler de 1977. En particular, gran parte del material presentado en los Capítulos 37, 38 y 39 está inspirado en este libro. La obra de Press *et al.* es también un compendio de métodos numéricos muy útiles, con sus implementaciones.

La otra referencia fundamental para esta sección es el tomo segundo del completo tratado de D. E. Knuth *The Art of Computer Programming*. Este autor utiliza el término «seminumérico» para describir los algoritmos que están situados como interfaz entre el cálculo numérico y el simbólico, tales como la generación de números aleatorios y la aritmética polinómica. El Volumen 2 de Knuth trata en gran profundidad, entre otros muchos temas, el material de los Capítulos 1, 3 y 4.

El libro de Borodin y Munro de 1975 es una referencia suplementaria para el método de multiplicación de matrices de Strassen, y presenta un tratamiento general de los algoritmos aritméticos desde el punto de vista de la complejidad de cálculo.

Muchos de los algoritmos que se han considerado en esta sección (y otros muchos, principalmente los métodos simbólicos mencionados en el Capítulo 39) se encuentran en un sistema informático denominado *Mathematica*, el más conocido de los diversos sistemas de «matemáticas simbólicas» que se han desarrollado en los últimos años. Los sistemas como *Mathematica* se han hecho indispensables entre los científicos y matemáticos para el análisis matemático de un gran número de aplicaciones.

- A. Borodin e I. Munro, *The Computational Complexity of Algebraic and Numerical Problems*, American Elsevier, New York, 1975.
- S. Conte y C. de Boor, *Elementary Numerical Analysis*, McGraw-Hill, New York, 1980.
- G. E. Forsythe, M. A. Malcomb y C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
- D. E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, (2.^a ed.), 1981.
- W. H. Press, B. P. Flannery, S. A. Teukolsky y W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 1986.
- S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison Wesley, Reading, MA, 1988.

Temas avanzados

Algoritmos paralelos

Los algoritmos que se han estudiado hasta aquí son, en su mayor parte, muy robustos en su capacidad de aplicación. La mayoría de los métodos que se han considerado son de hace una década o más, y han sobrevivido a muchos cambios radicales en el hardware y el software. Los nuevos diseños de hardware y las nuevas capacidades de software pueden tener un gran impacto en algoritmos específicos, pero los buenos algoritmos sobre viejas máquinas son, por lo general, buenos algoritmos en las nuevas máquinas.

Una razón para esto es que la organización fundamental de las computadoras convencionales ha cambiado muy poco a través de los años. El diseño de la mayoría de las computadoras ha estado guiado por el mismo principio básico, desarrollado por J.von Neumann en los primeros tiempos de la informática moderna. Cuando se habla del *modelo de von Neumann* se hace referencia a una visión de la informática en la que instrucciones y datos están almacenados en la misma memoria y un procesador sencillo toma las instrucciones de la memoria, y las ejecuta (operando sobre los datos), una a una. Se han desarrollado mecanismos elaborados para hacer las computadoras más baratas, rápidas, pequeñas (físicamente) y grandes (lógicamente), pero la arquitectura de la mayoría de los sistemas informáticos se puede considerar como variaciones sobre el tema de von Neumann.

Sin embargo, los últimos cambios radicales en el coste de los componentes han hecho que sea práctico considerar máquinas de tipos totalmente diferentes, capaces de ejecutar un gran número de instrucciones en un instante, o en las que las instrucciones están «integradas» con el objetivo concreto de resolver un problema único, o en las que un gran número de pequeños procesadores pueden cooperar para resolver el mismo problema. En pocas palabras, en lugar de tener una máquina que ejecuta una sola instrucción en cada momento, se puede pensar en una que ejecute simultáneamente gran número de acciones. En este capítulo se considerará el efecto potencial de tales ideas en algunos de los problemas y algoritmos que se han estudiado. En particular, se estudiarán dos va-

riantes de máquinas que son adecuadas para desarrollar algoritmos paralelos: la *mezcla perfecta* y los *arrays sistólicos*.

Aproximaciones generales

Algunos algoritmos fundamentales se utilizan con tanta frecuencia y para problemas tan grandes que siempre hay necesidad de ejecutarlos en máquinas más grandes y más rápidas. Una consecuencia de esto ha sido la aparición de una serie de «supercomputadoras» que utilizan la tecnología más reciente; éstas constituyen algunas derogaciones del concepto de la máquina de von Neumann, pero siguen estando diseñadas para propósito general y para que sean útiles a todos los programas. La estrategia común para utilizar tales máquinas en el tipo de problema que se ha estudiado consiste en comenzar con los mejores algoritmos sobre máquinas convencionales y adaptarlos a las características particulares de la nueva máquina. Esta aproximación estimula claramente la persistencia de los viejos algoritmos y de las antiguas arquitecturas en las nuevas máquinas.

Los microprocesadores con gran capacidad de cálculo se han hecho muy baratos. Una aproximación evidente es tratar de utilizar conjuntamente un gran número de estos microprocesadores para resolver un gran problema. Algunos algoritmos se adaptan bien a ser «distribuidos» de esta forma; otros simplemente no son apropiados para este tipo de implementación.

El desarrollo de procesadores de poco coste y relativamente potentes ha llevado también a la aparición de poderosas herramientas de propósito general a utilizar en el diseño y construcción de los nuevos procesadores. Esto ha llevado a un aumento en la actividad de desarrollo de máquinas de propósito específico para problemas particulares. Si ninguna máquina está particularmente bien diseñada para ejecutar un algoritmo importante, entonces ¡se puede diseñar y construir una que lo sea! Se pueden diseñar y construir máquinas apropiadas para muchos problemas e integrarlas en un chip de circuito (a gran escala de integración).

El hilo conductor de todas estas aproximaciones es el *parallelismo*: se trata de ahorrar tiempo ejecutando en cada instante tantas acciones diferentes como sea posible. Esto pudiera llevar a un caos si no se hace de una manera ordenada. Posteriormente se tratarán dos ejemplos que ilustran algunas técnicas que permiten lograr un alto grado de parallelismo en algunas clases específicas de problemas. La idea es suponer que se dispone no de uno, sino de M procesadores en los que se pueden ejecutar los programas. De esta forma, si las cosas salen bien, se puede esperar que los programas se ejecuten M veces más rápido que antes.

Existen varios problemas inmediatos al intentar que M procesadores trabajen conjuntamente para resolver un mismo problema. El más importante es que se deben comunicar de alguna manera: deben estar conectados entre sí y tener

mecanismos específicos para enviar y recibir datos a través de estas conexiones. Además, existen limitaciones físicas en el tipo de interconexión permitida. Por ejemplo, suponiendo que los «procesadores» son chips de circuitos integrados (que pueden contener ahora más circuitos que una computadora del pasado) que tengan, por ejemplo, 32 patillas para la interconexión, aun en el caso de tener 1.000 procesadores, se podría conectar cada uno de ellos con un máximo de otros 32. La selección de cómo interconectar los procesadores es fundamental en el paralelismo. Más aún, es importante recordar que esta decisión debe hacerse con anterioridad: un programa puede cambiar la forma como hace las cosas dependiendo de un caso particular del problema que está resolviendo, pero una máquina, en general, no puede cambiar la forma como están conectados entre sí sus componentes.

Esta visión general del paralelismo en términos de procesadores independientes con algunos modelos de interconexiones fijas se aplica a cada uno de los tres dominios descritos anteriormente: una supercomputadora tiene procesadores muy específicos y configuraciones de interconexión que están integrados en su arquitectura (y afectan a muchos aspectos de su rendimiento); los miprocesadores interconectados hacen ellos mismos intervenir a un número relativamente pequeño de procesadores potentes con interconexiones simples; y los circuitos a muy gran escala de integración (VLSI) hacen ellos mismos intervenir a un gran número de procesadores simples (elementos de los circuitos) con interconexiones complejas.

Se han estudiado extensivamente muchos otros enfoques del paralelismo desde von Neumann, y con renovado interés, puesto que cada vez aparecen más procesadores de bajo coste. El poder tratar todos los aspectos involucrados se saldría del alcance de este libro. En lugar de ello, se considerarán dos máquinas específicas que se han propuesto para resolver algunos problemas familiares, que ilustran los efectos de la arquitectura de la máquina en el diseño de algoritmos y viceversa. Aquí se constatará una cierta simbiosis: parece inconcebible diseñar una nueva computadora si no se tiene alguna idea de en qué se utilizará, y se desea utilizar las mejores computadoras disponibles para ejecutar los algoritmos fundamentales más importantes.

Mezcla perfecta

Para ilustrar algunos de los aspectos implicados en la implementación física (y no lógica) de algoritmos, se estudiará un método interesante de fusión adaptado a una implementación por hardware. Como se verá, se puede desarrollar el mismo método general para diseñar una «máquina algoritmo» que incorpore una configuración fundamental de interconexión para llevar a cabo operaciones paralelas de M procesadores para resolver varios problemas además del de fusión.

Como se mencionó anteriormente, una diferencia fundamental entre escri-

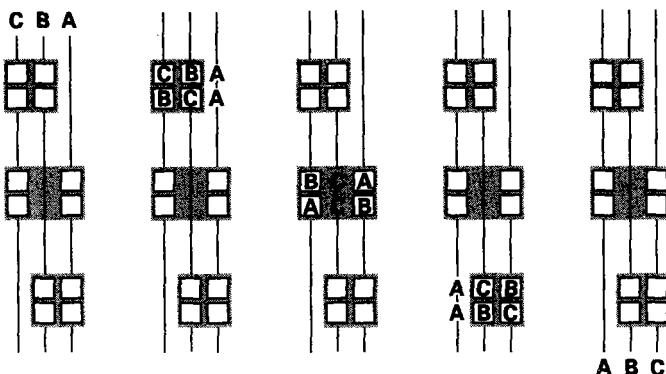


Figura 40.1 Máquina que ordena tres elementos.

bir un programa para resolver un problema y diseñar una máquina es que un programa puede *adaptar* su comportamiento a un elemento particular del problema que se está resolviendo, mientras que la máquina debe «programarse» por adelantado para hacer siempre la misma secuencia de operaciones. Para ver la diferencia considérese el primer programa de ordenación estudiado en el Capítulo 8, ordenar3. Independientemente de los tres números que aparecen en los datos, el programa siempre lleva a cabo la misma secuencia de las tres operaciones fundamentales de «comparar-intercambiar». Ningún otro algoritmo de ordenación de los que se han estudiado tiene esta propiedad. Todos ellos llevan a cabo una serie de comparaciones que dependen del resultado de comparaciones anteriores, y por ello presentan serios problemas en las implementaciones de hardware.

Concretamente, si se dispone de un circuito con dos conectores de entrada y dos de salida, que puede comparar dos números de la entrada e intercambiarlos si es necesario en la salida, se pueden conectar tres circuitos de este tipo, como se muestra en la Figura 40.1, para obtener una máquina de ordenación con tres entradas (en la parte superior de la figura) y tres salidas (en la inferior). Aquí, la primera «caja» intercambia C y B, la segunda B y A, y la tercera C y B para dar el resultado ordenado. La máquina ordena cualquier permutación de las entradas (como lo hizo ordenar3).

Por supuesto, se deben regular muchos detalles antes de que se pueda construir una tal máquina de ordenación basada en este esquema. Por ejemplo, no se ha especificado el método de codificación de las entradas. Una forma sería considerar cada conexión del diagrama como un «bus» lo suficientemente ancho como para transportar los datos en paralelo; otra forma es hacer que los «comparadores-intercambiadores» lean sus entradas un bit a la vez en una sola conexión (primero el bit más significativo). También se ha dejado sin especificar la sincronización: se deben incluir mecanismos para asegurar que no se lleva a cabo una «comparación-intercambio» antes de que se haya leído toda la en-

A E G G I M N R	A B E E I M N R
A B E E L M P X	A E G G L M P X
A B E E	A B E E
I M N R	A E G G
A E G G	I M N R
L M P X	L M P X
A B	A B
E E	A E
A E	E E
G G	G G
I M	I M
N R	L M
L M	N R
P X	P X
A	A
B	A
A	B
E	E
E	E
G	G
G	G
I	I
M	T
L	M
M	M
N	N
R	P
P	R
X	X

Figura 40.2 Fusión por división e intercalado.

trada. En este libro, evidentemente no será posible profundizar en estas cuestiones de diseño de circuitos; en su lugar se centrará la atención en los aspectos de alto nivel relativos a la interconexión de procesadores simples, como los comparadores-intercambiadores, para la solución de problemas más grandes.

Para comenzar, se considera un algoritmo para la fusión de dos archivos ordenados, utilizando una secuencia de operaciones «comparar-intercambiar», que es independiente de los números a fusionar y que es conveniente para implementaciones de hardware. La Figura 40.2 muestra el funcionamiento de este método al fusionar dos archivos ordenados en uno solo.

En primer lugar se escribe un archivo debajo del otro, y luego se comparan los elementos verticalmente adyacentes y se intercambian si es necesario para poner el mayor debajo del menor. A continuación se divide cada línea por la mitad y se intercalan las mitades, llevando a cabo las mismas operaciones «comparar-intercambiar» sobre los números de la segunda y la tercera línea. (Obsérvese que las operaciones que implican a las otras dos líneas no son ne-

cesarias debido a la ordenación previa.) Esto deja las filas y las columnas ordenadas. Este hecho es una propiedad fundamental del método, que el lector podría desear comprobar que es cierta, aunque una prueba rigurosa es un ejercicio más delicado de lo que parece.

Entonces resulta que en cada iteración se preserva el orden por la propia operación: dividir cada línea en dos, intercalar las mitades y hacer la «comparación-intercambio» entre los elementos verticalmente adyacentes que están en líneas diferentes. Cada paso duplica el número de filas, divide por la mitad el número de columnas, y mantiene las filas y las columnas ordenadas. Al principio hay 16 columnas y 1 fila, luego 8 columnas y 2 filas, luego 4 columnas y 4 filas, luego 2 columnas y 8 filas, y finalmente 16 filas y 1 columna, que está ordenada.

Propiedad 40.1 *La fusión de dos archivos ordenados de N elementos se puede llevar a cabo en alrededor de $\lg N$ etapas paralelas.*

Si $N = 2^n$, el método descrito lleva evidentemente n etapas, cada una de las cuales necesita menos de $N/2$ comparaciones independientes. Para probar que este método ordena, se necesita demostrar que las columnas permanecen ordenadas: esto se deja como ejercicio, como se mencionó anteriormente. Se pueden manejar otros tamaños de forma directa añadiendo claves ficticias.■

La operación básica «dividir cada línea por la mitad e intercalar las mitades» de la descripción anterior es fácil de visualizar sobre el papel, pero ¿cómo se traduce en las conexiones de una máquina? Hay una respuesta sorprendente y elegante a esta pregunta, que se deduce directamente escribiendo las tablas de forma diferente. En lugar de escribirlas hacia abajo en forma bidimensional, se escribirán como una simple lista (unidimensional) de números, organizada por *orden de columnas*: primero se colocan los elementos de la primera columna, luego los de la segunda, etc. Puesto que las ordenaciones comparar-intercambiar se hacen solamente entre los elementos verticalmente adyacentes, esto significa que cada estado implica un grupo de cajas comparar-intercambiar, conectadas entre sí según la operación de «dividir e intercalar» que se necesita para llevar los elementos a las cajas.

Esto conduce a la Figura 40.3, que corresponde precisamente a la descripción a base de las tablas anteriores, con la excepción de que se escribieron todas por orden de columnas (incluyendo una tabla inicial de 1*16 con un archivo, y luego la otra). Se aconseja al lector que compruebe la correspondencia entre este diagrama y las tablas dadas anteriormente. Las cajas comparar-intercambiar se han dibujado explícitamente, y se han representado las líneas que muestran cómo se mueven los elementos en la operación «dividir e intercalar». Sorprendentemente en esta representación cada operación «dividir e intercalar» se reduce exactamente a la misma configuración de interconexión. Este modelo se denomina la *mezcla perfecta* porque las conexiones están intercaladas, de la misma forma que se mezclan las cartas de una baraja.

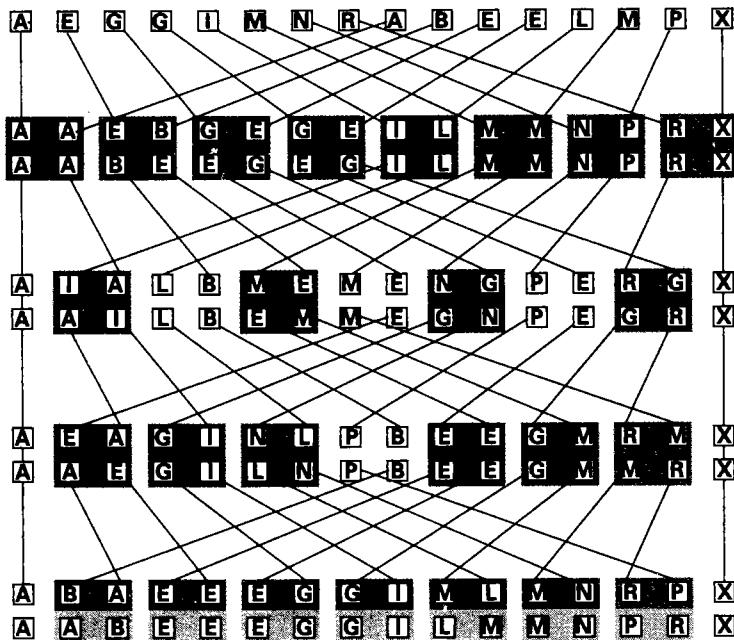


Figura 40.3 Fusión par-impar con mezcla perfecta.

Este método fue llamado *fusión par-impar* por K.E.Batcher, quien lo inventó en 1968. La característica esencial del método es que todas las operaciones comparar-intercambiar de cada etapa se pueden realizar en paralelo. Como afirma la propiedad 40.1, esto es significativo porque demuestra claramente que dos archivos se pueden fusionar en $\log N$ etapas paralelas (el número de filas de la tabla se divide por dos en cada paso), utilizando menos de $N \log N$ cajas de comparación-intercambio. A partir de la descripción anterior, esto puede parecer como un resultado directo; en realidad, el problema de encontrar una tal máquina ha constituido un desafío para los investigadores desde hace bastante tiempo.

Batcher desarrolló también un algoritmo de fusión bastante emparentado (aunque más difícil de entender), la *fusión bitónica*, que conduce a una máquina aún más simple, mostrada en la Figura 40.4. Este método se puede describir en términos de la operación «dividir e intercalar» sobre tablas como la anterior, excepto que se comienza con el segundo archivo ordenado a la *inversa* y haciendo siempre las comparaciones-intercambios entre elementos verticalmente adyacentes que provienen de las *mismas* líneas. No se va a intentar demostrar la validez de este método: el interés en él radica en que elimina la molesta característica de la fusión par-impar de que las cajas comparación-

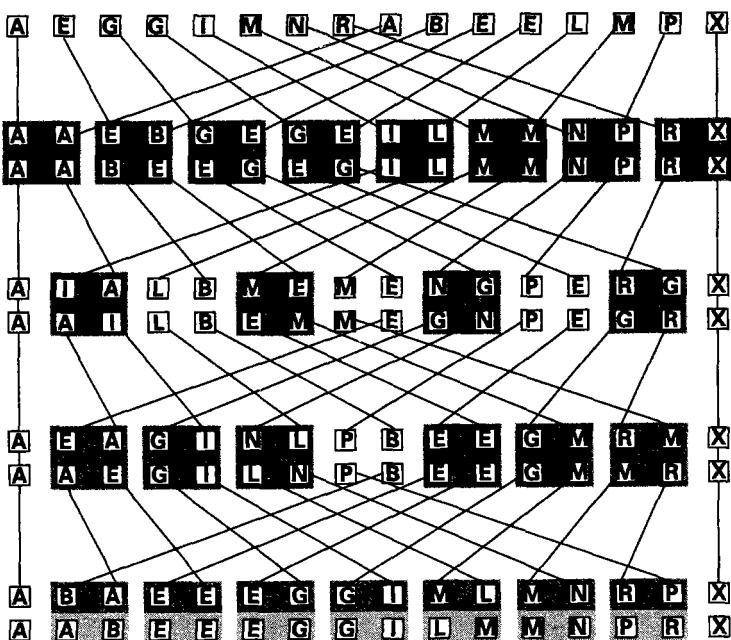


Figura 40.4 Fusión bitónica con mezcla perfecta.

intercambio de la primera etapa se desplazan una posición en las etapas siguientes. Como se muestra en la Figura 40.4, cada etapa de la fusión bitónica tiene exactamente el mismo número de comparadores, exactamente en las mismas posiciones.

Ahora hay regularidad no sólo en las interconexiones, sino en las posiciones de las cajas de comparación-intercambio. Hay más cajas que en la fusión par-impar, pero éste no es un problema, puesto que hay el mismo número de pasos paralelos. La importancia de este método es que conduce directamente a una forma de hacer la fusión utilizando sólo N cajas de comparación-intercambio. La idea es reducir simplemente las filas de la tabla anterior a un par de ellas y obtener, por tanto, una máquina conectada en circuito cerrado como se muestra en la Figura 40.5, que puede hacer $\log N$ «ciclos» comparación-intercambio-mezcla, uno por cada una de las etapas de la figura.

Es preciso observar cuidadosamente que esto no es un comportamiento paralelo «ideal»: puesto que se pueden fusionar entre sí dos archivos de N elementos, utilizando un único procesador, en un número de pasos proporcional a N , se puede esperar hacer la fusión en un número constante de pasos, utilizando N procesadores. En este caso, sin embargo, se ha demostrado que este ideal es imposible de lograr, y que la máquina anterior logra, el mejor funcionamiento paralelo posible para la fusión utilizando cajas comparación-intercambio.

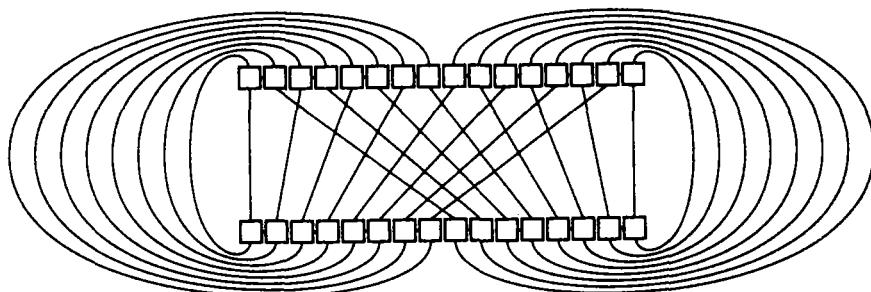


Figura 40.5 Una máquina de mezcla perfecta.

La configuración de interconexión de la mezcla perfecta es apropiada para un gran número de problemas. Por ejemplo, si se tiene una matriz cuadrada de $2^n * 2^n$ ordenada por filas, se podría obtener su traspuesta (ordenada por columnas) en n mezclas perfectas. Entre los ejemplos más importantes se incluyen la transformada rápida de Fourier (que se examinará en el próximo capítulo), la ordenación (que se puede desarrollar aplicando recursivamente uno de los métodos anteriores), la evaluación de polinomios, y muchos otros. Cada uno de estos problemas se puede resolver utilizando una máquina de mezcla perfecta en circuito cerrado con las mismas interconexiones que las presentadas anteriormente, pero con diferentes (algunos más complicados) procesadores. Algunos investigadores incluso han sugerido utilizar la configuración de la mezcla perfecta para computadoras paralelas de «propósito general».

Arrays sistólicos

Un problema del modelo anterior es que los «hilos» utilizados para la interconexión son largos. Además, hay muchos entrecruzados: una mezcla con N conexiones implica un número de cruces proporcional a N^2 . Estas dos operaciones entrañan dificultades cuando se construye de hecho una máquina de mezcla perfecta: las conexiones muy grandes implican mayores tiempos de transmisión, y los cruces hacen que la interconexión sea cara y no conveniente.

Una forma natural de evitar ambos problemas es conectar sólo los procesadores que estén físicamente adyacentes. Como se dijo antes, se operan los procesadores sincrónicamente: en cada paso, cada procesador lee las entradas en sus vecinos, hace los cálculos y escribe las salidas en sus vecinos. Esto no es necesariamente restrictivo, y de hecho H. T. Kung mostró en 1978 que arrays de procesadores de este tipo permiten un empleo muy eficaz de los procesadores para algunos problemas fundamentales. A estos arrays los denominó *arrays sistólicos*.

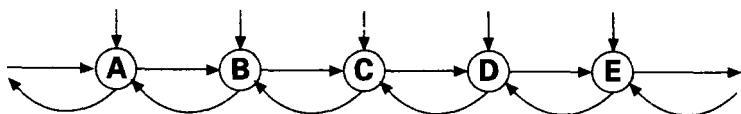


Figura 40.6 Un array sistólico.

tólicos porque la forma como los datos fluyen entre ellos recuerda a los latidos del corazón.

Como una aplicación representativa, considérese la utilización de los array sistólicos para la multiplicación de una matriz por un vector. En particular, se considera la operación de matrices

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}$$

Este cálculo se efectuará sobre una fila de procesadores simples, cada uno de los cuales tiene tres líneas de entrada y dos de salida, como se muestra en la Figura 40.6. Se utilizan cinco procesadores porque las entradas y salidas se efectuarán de forma secuencial muy precisa, como se describirá posteriormente.

En cada paso, cada procesador lee una entrada de la *izquierda*, una de lo *alto*, y una de la *derecha*; lleva a cabo un cálculo simple y escribe una salida a la *izquierda*, y una salida a la *derecha*. Concretamente, la salida *derecha* recibe todo lo que está en la entrada *izquierda* y la salida *alto* recibe los resultados calculados al multiplicar las entradas de la *izquierda* y de lo *alto*, y añadiendo la entrada de la *derecha*. Una característica crucial de los procesadores es que siempre llevan a cabo transformaciones dinámicas de las entradas hacia las salidas; nunca tienen que «recordar» los valores calculados. (Esto es también verdad para los procesadores de la máquina de mezcla perfecta.) Ésta es una regla impuesta por las restricciones de bajo nivel del diseño del hardware, puesto que la adición de una tal capacidad de «memoria» puede ser (relativamente) bastante cara.

El párrafo anterior da el «programa» para la máquina sistólica; para completar la descripción del cálculo se necesita especificar exactamente a qué ritmo se presentan los valores de entrada. Este tratamiento del tiempo es una característica esencial de la máquina sistólica, en marcado contraste con la máquina de la mezcla perfecta, donde todos los valores de entrada se presentan de una vez y todos los valores de salida están disponibles un momento después.

El plan general es presentar la matriz a través de las entradas *altas* de los procesadores, traspuesta y girada 45 grados, y el vector a través de la entrada *izquierda* del procesador A, para ser pasado a los otros procesadores. Los resultados intermedios se pasan de izquierda a derecha, con la salida apareciendo fi-

nalmente en la salida *izquierda* del procesador A. La sincronización específica para el ejemplo se muestra en la Figura 40.7.

El vector de entrada se presenta en la entrada *izquierda* del procesador A en los pasos 1, 3 y 5, y se pasa a la derecha a los otros procesadores en los pasos siguientes. La matriz de entrada se presenta a las entradas *altas* de los procesadores comenzando en el paso 3; los movimientos de izquierda a derecha por las diagonales de la matriz se presentan en los pasos sucesivos. El vector de salida aparece como la salida *izquierda* del procesador A en los pasos 6, 8 y 10. (En el diagrama, esto aparece como la entrada *derecha* de un procesador imaginario a la izquierda de A, que recogería la respuesta.)

El cálculo real se puede trazar siguiendo las entradas *derechas* (las salidas *izquierdas*) que se mueven de derecha a izquierda a través del array. Todos los cálculos producen un resultado cero hasta el paso 3; después de éste, el procesador C tiene 1 en sus entradas *izquierda* y *alta*, por lo que calcula el resultado 1, el cual se pasa como entrada *derecha* al procesador B para el paso 4. En esta etapa, el procesador B tiene valores no nulos para sus tres entradas, y calcula el valor 16, para pasarlo al procesador A para el paso 5. Mientras tanto, el procesador D calcula un valor 1 para que el procesador C lo utilice en el paso 5. En esta etapa, el procesador A calcula el valor 8, el cual se presenta como primer valor de salida en el paso 6; C calcula el valor 6 para que lo utilice B en el paso 6, y E calcula su primer valor no nulo (-1) para que lo utilice D en el paso 6. El cálculo del segundo valor de salida se completa por B en el paso 6 y se pasa a través de A hacia la salida en el paso 8, y el cálculo del tercer valor de salida se completa por C en el paso 7 y se pasa a través de B y A hacia la salida en el paso 10.

Una vez que se ha descrito el proceso a un nivel detallado se comprende mejor el método en un cierto nivel más alto. Los números de la parte intermedia de la Figura 40.7 (en rectángulos) son simplemente una copia de la matriz de entrada, traspuesta y girada como se necesita para su presentación como entradas *altas* de los procesadores. Si se observan los números de las correspondientes entradas *izquierdas* de los procesadores que reciben la matriz, se encuentran tres copias del vector de entrada, localizadas exactamente en las posiciones correctas y en el momento correcto para permitir la multiplicación por las filas de la matriz. Después, las salidas *derechas* de los procesadores muestran los resultados intermedios de cada multiplicación del vector de entrada por cada fila de la matriz. Por ejemplo, el producto del vector de entrada y de la fila media de la matriz necesita los cálculos parciales $1 * 1 = 1$, $1 + 1 * 5 = 6$, y $6 + (-2) * 2 = 2$ y éstos aparecen en las entradas 1 6 2 (leyendo diagonalmente, un paso abajo de la fila media de la matriz 1 1 -2). La máquina sistólica puede sincronizar las cosas de tal forma que cada elemento de la matriz «se encuentre» con el buen vector de entrada y el buen cálculo parcial en el nivel del procesador donde éste entra, por lo que se puede incorporar en el resultado intermedio.

El método se generaliza de forma evidente al producto de una matriz $N * N$ por un vector $N * 1$ utilizando $2N - 1$ procesadores en $4N - 2$ pasos. Esto es una aproximación a la situación ideal de tener a cada procesador haciendo un tra-

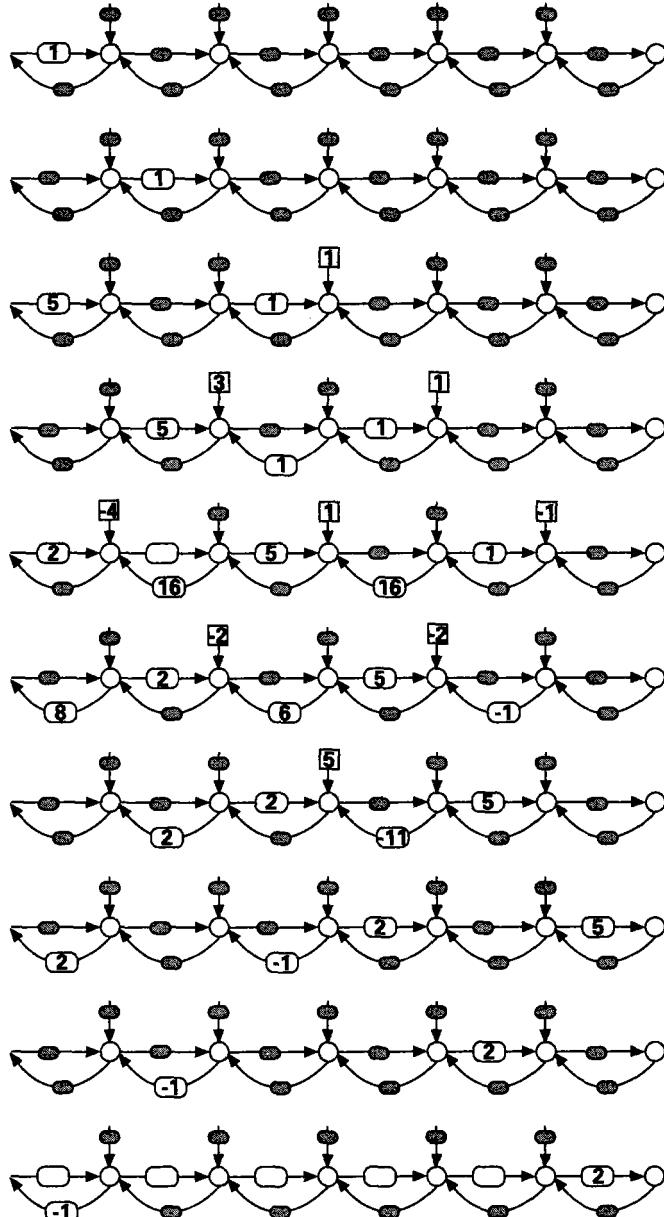


Figura 40.7 Multiplicación de una matriz por un vector con un array sistólico.

bajo útil en cada paso: un algoritmo cuadrático se reduce a un algoritmo lineal utilizando un número lineal de procesadores.

Como anteriormente, sólo se ha descrito un método general de cálculo en paralelo. Se necesita regular muchos detalles del diseño lógico antes de poder construir una tal máquina sistólica. Se puede apreciar en este ejemplo que los arrays sistólicos son a su vez simples y potentes. El vector de salida parece que sale por el borde ¡como por arte de magia! Sin embargo, cada procesador individual no hace más que llevar a cabo el cálculo simple antes descrito: la magia está en la interconexión y en la sincronización en las entradas.

Al igual que las máquinas de mezcla perfecta, los arrays sistólicos se pueden utilizar en muchos tipos de problemas diferentes, incluyendo el reconocimiento de cadenas y la multiplicación de matrices, entre otros. También, al igual que las máquinas de mezcla perfecta, algunos investigadores han sugerido utilizar estas configuraciones de interconexión para máquinas paralelas de «propósito general».

Perspectiva

El estudio de las máquinas de mezcla perfecta y de las máquinas sistólicas ilustra el hecho de que el diseño del hardware puede tener una influencia significativa en el diseño de algoritmos, y propone cambios que pueden proporcionar nuevos e interesantes algoritmos y nuevos desafíos para los diseñadores de algoritmos.

A pesar de que ésta es un área interesante y fructífera para la investigación futura, se debe concluir con unas serenas anotaciones. Primero, se necesita un gran esfuerzo de ingeniería para traducir esquemas generales de cálculo en paralelo, como los esbozados anteriormente en algoritmos reales de máquina con un buen rendimiento. Para muchas aplicaciones, el gasto en recursos que se necesita simplemente no se justifica, ya que una simple «máquina de algoritmos», consistente en un microprocesador convencional (no costoso), que ejecute un algoritmo convencional, lo haría bastante bien. Por ejemplo, si se tienen muchos elementos del mismo problema a resolver y varios microprocesadores para resolverlos, entonces se podría alcanzar un funcionamiento paralelo ideal teniendo a cada microprocesador (utilizando un algoritmo convencional) trabajando sobre un elemento particular del problema, sin conexión entre ellos. Si se tienen N archivos a ordenar y N procesadores disponibles para hacer la ordenación, ¿por qué no utilizar un procesador para cada ordenación, en lugar de tener a todos los N procesadores trabajando juntos sobre las N ordenaciones?

Es bastante difícil evaluar el impacto de las diferentes estrategias de cálculo paralelo sobre el rendimiento de los algoritmos. Todos los aspectos presentados en los Capítulos 6 y 7 se deben tener en consideración, con la complicación adicional de que la máquina en sí se convierte en una variable. La forma más fácil de comparar máquinas (y por supuesto la más comúnmente utilizada) puede

estar extremadamente distorsionada: ejecutar el mismo programa en ambas máquinas es exactamente lo que no hay que hacer si el algoritmo en cuestión depende mucho de la arquitectura de una de las máquinas y no de la otra. Las concordancias deben hacerlo mucho mejor que las discordancias. La comparación conveniente de dos máquinas debe hacerse basándose en el *problema a resolver* y buscando el mejor algoritmo para cada problema en cada máquina. ¿La nueva arquitectura permite resolver un problema que no puede ser resuelto con la más antigua?

Las técnicas como las presentadas en este capítulo se justifican sólo en aplicaciones con requisitos muy especiales de tiempo y espacio. Del estudio de diversos esquemas de cálculo paralelo y sus efectos sobre el rendimiento de algunos algoritmos, se puede esperar el desarrollo de computadoras paralelas de propósito general que mejorarán el funcionamiento de una gran variedad de algoritmos.

Ejercicios

1. Esbozar dos posibles formas de utilizar el paralelismo en el Quicksort.
2. Demostrar que el método de «dividir e intercalar» preserva el orden de las columnas ordenadas.
3. Escribir un programa convencional en C++ para fusionar archivos utilizando el método bitónico de Batcher.
4. Escribir un programa convencional en C++ para fusionar archivos utilizando el método bitónico de Batcher, pero sin hacer realmente ninguna mezcla.
5. ¿Cuántas mezclas perfectas restaurarían el orden inicial de todos los elementos de un array de tamaño 2^n ?
6. Dibujar una tabla como la de la Figura 40.7 para ilustrar el funcionamiento del multiplicador sistólico de matriz vector para el problema siguiente:

$$\begin{pmatrix} 2 & 1 & 4 \\ 3 & 0 & 1 \\ 1 & -1 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -1 \end{pmatrix}.$$

7. Escribir un programa convencional en C++ que simule la operación del array sistólico para multiplicar una matriz $N * N$ por un vector $N * 1$.
8. Mostrar cómo utilizar un array sistólico para hallar la traspuesta de una matriz.
9. ¿Cuántos procesadores y cuántos pasos se necesitan para una máquina sistólica capaz de multiplicar una matriz de $M * N$ por un vector de $N * 1$?
10. Encontrar un modelo paralelo simple para la multiplicación de matriz vector utilizando procesadores que tengan la capacidad de «recordar» los valores calculados.

La transformada rápida de Fourier

Uno de los algoritmos aritméticos más ampliamente utilizados es la *transformada rápida de Fourier*, un medio eficaz de ejecutar un cálculo matemático básico y de frecuente empleo. La transformada de Fourier es de importancia fundamental en el análisis matemático y ha sido objeto de numerosos estudios. La aparición de un algoritmo eficaz para esta operación fue una piedra angular en la historia de la informática.

Las aplicaciones de la transformada rápida de Fourier son múltiples. Es la base de muchas operaciones fundamentales del procesamiento de señales, donde tiene amplia utilización. Además, como se verá, proporciona un medio conveniente para mejorar el rendimiento de los algoritmos para un conjunto de problemas aritméticos comunes. Describir la base matemática de la transformada de Fourier o resumir sus muchas aplicaciones está fuera del alcance de este libro; el objetivo de este capítulo es observar las características de un algoritmo fundamental para esta transformación dentro del contexto de alguno de los algoritmos que se han venido estudiando.

En particular, se examinará cómo utilizar el algoritmo para reducir sustancialmente el tiempo necesario para una multiplicación de polinomios, problema que se estudió en el Capítulo 36. Sólo se necesitan algunos resultados elementales del análisis complejo para mostrar cómo se puede utilizar la transformada de Fourier para multiplicar polinomios, y es posible apreciar el algoritmo de la transformada rápida de Fourier sin comprender completamente los fundamentos matemáticos en que se basa. Este algoritmo aplica la técnica de divide y vencerás de forma muy similar a otros algoritmos que ya se han visto.

Evaluar, multiplicar, interpolar

La estrategia general del método mejorado para la multiplicación de polinomios que se va a examinar se aprovecha del hecho de que un polinomio de grado $N - 1$ está determinado por completo por su valor en N puntos diferentes. Al multiplicar dos polinomios de grado $N - 1$ entre sí se obtiene un polinomio de grado $2N - 2$; si se puede determinar los valores de un polinomio en $2N - 1$ puntos, entonces éste está completamente determinado. Pero es posible determinar el valor del resultado en cualquier punto evaluando simplemente en ese punto los dos polinomios a multiplicar y multiplicando los valores obtenidos.

Esto conduce al siguiente esquema general para la multiplicación de dos polinomios de grado $N - 1$:

Evaluar los polinomios de datos en $2N - 1$ puntos.

Multiplicar los dos valores obtenidos en cada punto.

Interpolar para encontrar el único polinomio resultante que toma los valores obtenidos en los puntos dados.

Por ejemplo, para calcular $r(x) = p(x)q(x)$ con $p(x) = 1 + x + x^2$ y $q(x) = 2 - x + x^2$, se puede evaluar $p(x)$ y $q(x)$ en cinco puntos cualesquiera, por ejemplo $-2, -1, 0, 1, 2$, para obtener los valores

$$\begin{aligned}[p(-2), p(-1), p(0), p(1), p(2)] &= [3, 1, 1, 3, 7], \\ [q(-2), q(-1), q(0), q(1), q(2)] &= [8, 4, 2, 2, 4].\end{aligned}$$

Multiplicando éstos entre sí término a término se obtienen suficientes valores para que el producto polinómico,

$$r(-2), r(-1), r(0), r(1), r(2)] = [24, 4, 2, 6, 28],$$

esté totalmente determinado por interpolación. Por la fórmula de Lagrange

$$\begin{aligned}r(x) &= 24 \frac{x+1}{-2+1} \frac{x-0}{-2-0} \frac{x-1}{-2-1} \frac{x-2}{-2-2} \\ &\quad + 4 \frac{x+2}{-1+2} \frac{x-0}{-1-0} \frac{x-1}{-1-1} \frac{x-2}{-1-2} \\ &\quad + 2 \frac{x+2}{0+2} \frac{x+1}{0+1} \frac{x-1}{0-1} \frac{x-2}{0-2} \\ &\quad + 6 \frac{x+2}{1+2} \frac{x+1}{1+1} \frac{x-0}{1-0} \frac{x-2}{1-2} \\ &\quad + 28 \frac{x+2}{2+2} \frac{x+1}{2+1} \frac{x-0}{2-0} \frac{x-1}{2-1},\end{aligned}$$

se obtiene, después de simplificar, el resultado

$$r(x) = 2 + x + 2x^2 + x^4.$$

Como se presentó anteriormente, este método no es un algoritmo atractivo para la multiplicación de polinomios puesto que las mejores técnicas con que se cuenta hasta ahora para la evaluación (aplicación repetida del método de Horner) e interpolación (fórmula de Lagrange) necesitan N^2 operaciones. Sin embargo, hay una cierta esperanza de encontrar un algoritmo mejor porque el método es válido para cualquier elección de $2N - 1$ puntos totalmente distintos, y es razonable esperar que la evaluación y la interpolación sean más fáciles para ciertos conjuntos de puntos que para otros.

Raíces complejas de la unidad

Resulta que los puntos más convenientes a utilizar en una interpolación y evaluación de polinomios son los números complejos, de hecho, un conjunto particular de ellos denominados *raíces complejas de la unidad*.

Es necesaria una breve revisión de algunos temas sobre el análisis complejo. El número $i = \sqrt{-1}$ es un número *imaginario*: aunque $\sqrt{-1}$ no tiene significado como número real, es conveniente darle un nombre, i , y llevar a cabo operaciones algebraicas con él, reemplazando i^2 por -1 cada vez que aparezca. Un *número complejo* consta de dos partes, una real y una imaginaria; suele escribirse como $a + bi$, donde a y b son reales. Para multiplicar dos números complejos, se aplican las reglas normales, pero reemplazando i^2 por -1 cada vez que aparezca. Por ejemplo,

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i.$$

Algunas veces la parte real o imaginaria puede desaparecer cuando se lleva a cabo una multiplicación compleja. Por ejemplo,

$$(1 - i)(1 - i) = -2i,$$

$$(1 + i)^4 = -4,$$

$$(1 + i)^8 = 16.$$

Dividiendo la última ecuación por $16 = \sqrt{2}^8$, se ve que

$$\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}} \right)^8 = 1.$$

En general, hay muchos números complejos que dan como resultado 1 al elevarlos a una potencia. Éstos son los denominados raíces complejas de la unidad. De hecho, para cada N hay exactamente N números complejos z con $z^N = 1$. Uno de ellos w_N , se denomina *la raíz N-ésima principal de la unidad*; los otros se obtienen elevando w_N a la k -ésima potencia, para $k = 0, 1, 2, \dots, N - 1$. Por ejemplo se pueden obtener las ocho raíces de la unidad de la siguiente forma:

$$w_8^0, w_8^1, w_8^2, w_8^3, w_8^4, w_8^5, w_8^6, w_8^7.$$

La primera raíz, w_N^0 , es 1 y la segunda, w_N^1 , es la raíz principal. También, para N par, la raíz $w_N^{N/2}$ es -1 (porque $(w_N^{N/2})^2 = 1$). Los valores exactos de la raíz no son importantes por el momento. No se hará más que utilizar propiedades simples que se derivan fácilmente de la propiedad fundamental de que la N -ésima potencia de cualquier raíz N -ésima de la unidad debe ser 1.

Evaluación de las raíces de la unidad

La base de la implementación será un procedimiento para evaluar un polinomio de grado $N - 1$ en las N raíces de la unidad. Esto es, este procedimiento transforma los N coeficientes que definen al polinomio en los N valores obtenidos al evaluarlo en las N raíces de la unidad.

Aunque quizás esto no corresponda exactamente con lo que se quiere, puesto que para el primer paso del procedimiento de multiplicación de polinomios se necesita evaluar polinomios de grado $N - 1$ en $2N - 1$ puntos, en realidad, esto no es un problema, ya que se puede considerar un polinomio de grado $N - 1$ como un polinomio de grado $2N - 2$ con $N - 1$ coeficientes (los de los términos de mayor grado) que son cero.

El algoritmo que se utilizará para evaluar un polinomio de grado $N - 1$ en N puntos simultáneamente se basará en la simple estrategia de divide y vencerás. En lugar de estar dividiendo los polinomios por la mitad (como en el algoritmo de multiplicación del Capítulo 4) se dividirán en dos partes colocando alternativamente términos en cada parte. Esta división se puede expresar fácilmente en términos de polinomios que tienen la mitad de coeficientes. Por ejemplo, para $N=8$, la reorganización de términos es la siguiente:

$$\begin{aligned} p(x) &= p_0 + p_1x + p_3x^3 + p_4x^4 + p_5x^5 + p_6x^6 + p_7x^7 \\ &= (p_0 + p_2x^2 + p_4x^4 + p_6x^6) + x(p_1 + p_3x^2 + p_5x^4 + p_7x^6) \\ &\equiv p_e(x^2) + xp_0(x^2). \end{aligned}$$

Las N raíces de la unidad se adaptan bien a esta descomposición porque, si se eleva al cuadrado una raíz de la unidad, se obtiene otra raíz de la unidad. De

hecho, se puede ir aún más lejos: para N par, si se eleva al cuadrado una raíz N -ésima de la unidad, se obtiene la $\frac{1}{2}N$ -ésima potencia (un número que elevado a la $\frac{1}{2}N$ -ésima potencia vale 1). Esto es exactamente lo que se necesita para hacer que funcione el método de divide y vencerás. Para evaluar un polinomio de N coeficientes en N puntos, se divide en dos polinomios de $\frac{1}{2}N$ coeficientes. Estos últimos necesitan ser evaluados sólo en $\frac{1}{2}N$ puntos (las $\frac{1}{2}N$ raíces de la unidad) para obtener los valores necesarios para la evaluación completa.

Para ver esto con más claridad, considérese la evaluación de un polinomio $p(x)$ de grado 7 en las ocho raíces de la unidad

$$W_8 : w_8^0, w_8^1, w_8^2, w_8^3, w_8^4, w_8^5, w_8^6, w_8^7.$$

Puesto que $w_8^4 = -1$, esto es lo mismo que

$$W_8 : w_8^0, w_8^1, w_8^2, w_8^3, -w_8^0, -w_8^1, -w_8^2, -w_8^3.$$

Elevando al cuadrado cada uno de los términos de esta serie se obtienen dos copias de la serie $\{W_4\}$ de las raíces cuartas de la unidad:

$$W_8^2 : w_4^0, w_4^1, w_4^2, w_4^3, w_4^0, w_4^1, w_4^2, w_4^3.$$

Ahora, la ecuación

$$p(x) = p_e(x^2) + xp_0(x^2)$$

indica inmediatamente cómo evaluar $p(x)$ en las ocho raíces de la unidad a partir de estas series. Primero se evalúan $p_e(x)$ y $p_0(x)$ en las raíces cuartas de la unidad y luego se sustituye cada una de las raíces octavas de la unidad de x en la ecuación anterior, lo que necesita añadir el valor apropiado de p_e al producto del valor apropiado p_0 y la raíz octava de la unidad:

$$\begin{aligned} p(w_8^0) &= p_e(w_4^0) + w_8^0 p_0(w_4^0), \\ p(w_8^1) &= p_e(w_4^1) + w_8^1 p_0(w_4^1), \\ p(w_8^2) &= p_e(w_4^2) + w_8^2 p_0(w_4^2), \\ p(w_8^3) &= p_e(w_4^3) + w_8^3 p_0(w_4^3), \\ p(w_8^4) &= p_e(w_4^0) - w_8^0 p_0(w_4^0), \\ p(w_8^5) &= p_e(w_4^1) - w_8^1 p_0(w_4^1), \\ p(w_8^6) &= p_e(w_4^2) - w_8^2 p_0(w_4^2), \\ p(w_8^7) &= p_e(w_4^3) - w_8^3 p_0(w_4^3). \end{aligned}$$

En general, para evaluar $p(x)$ en las raíces N -ésimas de la unidad, se evalúan recursivamente $p_e(x)$ y $p_0(x)$ en las raíces $\frac{1}{2}N$ -ésimas de la unidad y se llevan a

cabo N multiplicaciones como las anteriores. Esto sólo es válido cuando N es par, y de este modo se supone que a partir de aquí N es una potencia de dos, por lo que se mantendrá par a lo largo de la recursión. Ésta se detiene cuando $N = 2$; y entonces hay que evaluar $p_0 + p_1x$ en 1 y -1, con los resultados $p_0 + p_1$ y $p_0 - p_1$.

Propiedad 41.1 *Un polinomio de grado $N - 1$ se puede evaluar en las raíces N -ésimas de la unidad con $N \lg N$ multiplicaciones.*

El número de multiplicaciones necesarias satisface la recurrencia fundamental de «divide y vencerás» $M(N) = 2M(N/2) + N$, cuya solución es $M(N) = N \lg N$ (fórmula 4 del Capítulo 6). Ésta es una mejora sustancial respecto al método de interpolación directa N^2 pero, por supuesto, sólo es válida en las raíces de la unidad. ■

Esto proporciona un método para transformar un polinomio de su representación convencional con N coeficientes a su representación en términos de sus valores en las raíces de la unidad. Esta conversión del polinomio de la primera representación a la segunda es la transformada de Fourier, y el procedimiento de cálculo recursivo que se ha descrito se denomina la transformada «rápida» de Fourier (FFT)[†]. (Estas mismas técnicas se aplican a funciones más generales que los polinomios. Concretamente, en este caso, se está haciendo la transformada «discreta» de Fourier.)

Interpolación en las raíces de la unidad

Ahora que se dispone de una forma rápida de evaluar polinomios en un conjunto específico de puntos, todo lo que se necesita es una forma también rápida de interpolar los polinomios en esos mismos puntos, y se tendrá un método rápido de multiplicación de polinomios. Sorprendentemente, esto funciona de modo que, para las raíces complejas de la unidad, ¡al ejecutar el programa de evaluación en un conjunto particular de puntos se hace la interpolación!

En el ejemplo, con $N = 8$, el problema de interpolación es encontrar el polinomio

$$r(x) = r_0 + r_1x + r_2x^2 + r_3x^3 + r_4x^4 + r_5x^5 + r_6x^6 + r_7x^7$$

que toma los valores

$$r(w_8^0) = s_0, \quad r(w_8^1) = s_1, \quad r(w_8^2) = s_2, \quad r(w_8^3) = s_3,$$

[†] Las siglas FFT están tomadas del inglés Fast Fourier Transformation. (*N. del T.*)

$$r(w_8^4) = s_4, \quad r(w_8^5) = s_5, \quad r(w_8^6) = s_6, \quad r(w_8^7) = s_7.$$

Cuando los puntos en cuestión son las raíces complejas de la unidad, es literalmente cierto que el problema de interpolación es el «inverso» del problema de evaluación. Si se tiene

$$s(x) = s_0 + s_1x + s_2x^2 + s_3x^3 + s_4x^4 + s_5x^5 + s_6x^6 + s_7x^7$$

entonces se pueden encontrar los coeficientes

$$r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7$$

evaluando simplemente el polinomio $s(x)$ en las inversas de la raíces complejas de la unidad

$$W_8^{-1}: w_8^0, w_8^{-1}, w_8^{-2}, w_8^{-3}, w_8^{-4}, w_8^{-5}, w_8^{-6}, w_8^{-7}.$$

Ésta es la misma serie que la de las raíces complejas de la unidad, pero en un orden diferente:

$$W_8^{-1}: w_8^0, w_8^7, w_8^6, w_8^5, w_8^4, w_8^3, w_8^2, w_8^1.$$

En otras palabras, se pueden utilizar exactamente las mismas rutinas para la interpolación que para la evaluación: sólo se necesita una simple restructuración de los puntos a evaluar.

La demostración de este hecho necesita algunas manipulaciones elementales con sumas finitas; los lectores no familiarizados con este tipo de manipulaciones pueden pasar directamente al final de este párrafo. Al evaluar $s(x)$ en la inversa de la t -ésima de las raíces N -ésimas de la unidad se obtiene

$$\begin{aligned} s(w_N^{-t}) &= \sum_{0 \leq j < N} s_j (w_N^{-t})^j \\ &= \sum_{0 \leq j < N} r_j (w_N^{-t})^j \\ &= \sum_{0 \leq j < N} \sum_{0 \leq i < N} r_i (w_N^{-t})^i (w_N^{-t})^j \\ &= \sum_{0 \leq j < N} r_j \sum_{0 \leq i < N} w_N^{j(i-t)} \\ &= \sum_{0 \leq j < N} r_j \sum_{0 \leq i < N} w_N^{j(i-t)} = N r_t. \end{aligned}$$

Casi todos los términos desaparecen en la última ecuación porque la suma interna es trivialmente N si $i = t$. Si $i \neq t$ entonces se hace

$$\sum_{0 \leq j < N} w_N^{j(i-t)} = \frac{w_N^{(i-t)N} - 1}{w_N^{(i-t)} - 1} = 0.$$

Obsérvese que aparece un factor multiplicativo de N . Éste es el «teorema de la inversión» para la transformada discreta de Fourier, que dice que el mismo método sirve para transformar el polinomio en ambas direcciones: pasando de su representación por coeficientes a la representación por sus valores en las raíces complejas de la unidad.

Propiedad 41.2 *Un polinomio de grado $N - 1$ se puede interpolar en las raíces N -ésimas de la unidad con alrededor de $N \lg N$ multiplicaciones.*

Los razonamientos matemáticos anteriores pueden parecer muy complicados, pero los resultados son bastante fáciles de aplicar: para interpolar un polinomio en las raíces N -ésimas de la unidad, se utiliza el mismo procedimiento que para la evaluación, utilizando los valores de interpolación como coeficientes, reorganizando después y dividiendo las respuestas por el factor apropiado.■

Implementación

Ahora se tienen todas las piezas del algoritmo de divide y vencerás para multiplicar dos polinomios utilizando sólo alrededor de $N \lg N$ operaciones. El esquema general es:

Evaluar los polinomios de entrada en las $(2N - 1)$ raíces de la unidad.

Multiplicar los dos valores obtenidos en cada punto.

Interpolar para encontrar el resultado evaluando el polinomio definido por los números obtenidos en las $(2N - 1)$ raíces de la unidad.

La descripción anterior se puede transformar directamente en un programa que utiliza un procedimiento para evaluar un polinomio de grado $N - 1$ en las raíces N -ésimas de la unidad. Toda la aritmética de este algoritmo es compleja, pero afortunadamente en C++ es fácil tener un tipo definido por el usuario para los números complejos. Esto se puede lograr gracias a la sobrecarga de los operadores aritméticos normales (éste es un ejemplo estándar en muchos libros de C++), por lo que no se oscurece innecesariamente el algoritmo. La implementación siguiente supone la existencia de un tal tipo *complejo*:

```
eval(p, salidaN, 0);
```

```

eval(q, salidaN, 0);
for (i = 0; i <= salidaN; i++) r[i] = p[i]*q[i];
eval(r, salidaN, 0);
for (i = 1; i <= N; i++)
{ t = r[i]; r[i] = r[salidaN+1-i]; r[salidaN+1-i] = t; }
for (i = 0; i <= salidaN; i++) r[i] = r[i]/(salidaN+1);

```

Este programa supone que a la variable global `salidaN` se le ha asignado el valor $2N-1$ y que `p`, `q` y `r` son arrays indexados entre 0 y $2N-1$ que contienen números complejos. Los dos polinomios a multiplicar, `p` y `q`, son de grado $N-1$, y los otros coeficientes de esos arrays se ponen a cero inicialmente. El procedimiento `eval` reemplaza los coeficientes de los polinomios que recibe como primer argumento por los valores obtenidos cuando el polinomio se evalúa en la raíces de la unidad. El segundo argumento especifica el grado de los polinomios (uno menos que el número de coeficientes y raíces de la unidad), y el tercer argumento se describe más tarde. El código anterior calcula el producto de `p` y `q` y coloca el resultado en `r`.

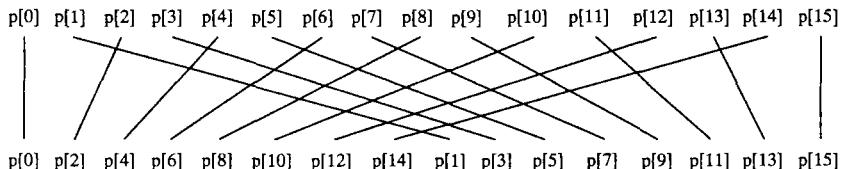


Figura 41.1 Desmezclado perfecto para la FFT.

Ahora sólo queda la implementación de `eval`. Como se ha visto antes, puede ser bastante engorroso implementar los programas recursivos que implican arrays. Esto provoca que en este algoritmo se evite el problema habitual de gestión de la memoria reutilizándola de forma ingeniosa. Lo que se quisiera tener es procedimientos recursivos que tomen como entrada un array de $N+1$ coeficientes contiguos y devuelvan los $N+1$ valores en el mismo array. Pero las etapas recursivas implican el procesamiento de dos arrays no contiguos: los coeficientes impares y los pares. El lector puede ya haber advertido que la mezcla perfecta del capítulo anterior es precisamente lo que se necesita aquí. Se pueden colocar los coeficientes impares en un subarray contiguo (la primera mitad) y los pares en un subarray contiguo (la segunda mitad) haciendo un «desmezclado perfecto» de la entrada, como muestra la Figura 41.1 para $N=15$.

Por supuesto, para hacer la implementación se necesitan los valores concretos de las raíces complejas de la unidad. Es bien conocido que

$$w_N^j = \cos\left(\frac{2\pi j}{N+1}\right) + i \sin\left(\frac{2\pi j}{N+1}\right);$$

estos valores se calculan fácilmente utilizando las funciones trigonométricas convencionales. En el siguiente programa, se supone que el array w contiene las $(\text{salidaN}+1)$ raíces de la unidad.

Esto deja la siguiente implementación de la FFT:

```
eval(complejo p[], int N, int k)
{
    int i, j;
    if (N == 1)
    {
        p0 = p[k]; p1 = p[k+1];
        p[k] = p0+p1; p[k+1] = p0-p1;
    }
    else
    {
        for (i = 0; i <= N/2; i++)
        {
            j = k+2*i;
            t[i] = p[j]; t[i+N/2] = p[j+1];
        }
        for (i = 0; i <= N; i++) p[k+i] = t[i];
        eval(p, N/2, k);
        eval(p, N/2, (k+1+N)/2);
        j = (salidaN+1)/(N+1);
        for (i = 0; i <= N/2; i++)
        {
            p0 = w[i*j] * p[k+(N/2)+1+i];
            t[i] = p[k+i] + p0;
            t[i+(N/2)+1] = p[k+i] - p0;
        }
        for (i = 0; i <= N; i++) p[k+i] = t[i];
    }
}
```

Este programa transforma el polinomio de grado N en el subarray $p[k], \dots, p[k+N]$ utilizando el método recursivo descrito anteriormente. (Por simplicidad, el código supone que $N+1$ es una potencia de dos, aunque esta dependencia no es difícil de eliminar.) Si $N = 1$, entonces se lleva a cabo el cálculo para evaluar en 1 y -1 , y en caso contrario el procedimiento comienza por mezclar, luego se llama a sí mismo recursivamente para transformar las dos mitades, y después combina los resultados de estos cálculos como se describió con anterioridad. Para

obtener las raíces de la unidad que se necesitan, el programa se sirve del array w en un intervalo determinado por la variable i . Por ejemplo, si salidaN es 15, las raíces cuartas de la unidad se encuentran en $w[0]$, $w[4]$, $w[8]$ y $w[12]$. Esto elimina la necesidad de volver a calcular las raíces de la unidad cada vez que se utilizan.

Propiedad 41.3 *Dos polinomios de grado N se pueden multiplicar con $2N\lg N + O(N)$ multiplicaciones de complejos.*

Este resultado se obtiene directamente de las propiedades 41.1 y 41.2.■

Como se mencionó al principio, el espectro de aplicación de la FFT es bastante mayor que lo que puede haberse mencionado aquí. El algoritmo se ha utilizado intensivamente y estudiado en una gran variedad de dominios. Sin embargo, los principios fundamentales de su funcionamiento en operaciones más avanzadas son los mismos que los del problema de multiplicación de polinomios presentado aquí. La FFT es un ejemplo clásico de aplicación del paradigma de diseño de algoritmos «divide y vencerás» para conseguir economías de cálculo verdaderamente significativas.

Ejercicios

1. ¿Cómo se podría mejorar el simple algoritmo de evaluar-multiplicar-interpolar para multiplicar entre sí dos polinomios $p(x)$ y $q(x)$ con raíces conocidas p_0, p_1, \dots, p_{N-1} y q_0, q_1, \dots, q_{N-1} ?
2. Encontrar un conjunto de N números reales en los que se pueda evaluar un polinomio de grado N utilizando muchas menos de N^2 operaciones.
3. Encontrar un conjunto de N números reales en los que se pueda interpolar un polinomio de grado N utilizando muchas menos de N^2 operaciones.
4. ¿Cuál es el valor de w_N^M para $M > N$?
5. ¿Merece la pena multiplicar polinomios dispersos utilizando la FFT?
6. La implementación de la FFT tiene tres llamadas a `eval`, exactamente igual que la multiplicación de polinomios del Capítulo 36 tiene tres llamadas a `mult`. ¿Por qué es más eficaz la implementación de la FFT?
7. Encontrar una forma de multiplicar entre sí dos números complejos utilizando menos de cuatro operaciones de multiplicación entera.
8. ¿Cuánta memoria sería necesaria para la FFT si no se puede evitar el problema de la gestión de memoria con la mezcla perfecta?
9. ¿Por qué no se puede utilizar una técnica como la de la mezcla perfecta para evitar problemas con arrays declarados dinámicamente en el procedimiento de multiplicación polinómica del Capítulo 36?
10. Escribir un programa eficaz para multiplicar un polinomio de grado N por un polinomio de grado M (no necesariamente potencias de dos).

Programación dinámica

El principio de *divide y vencerás* ha guiado el diseño de muchos de los algoritmos que se han estudiado hasta aquí: para resolver un gran problema, se divide en varios problemas más pequeños que se puedan resolver independiente-mente. En la programación dinámica este principio se lleva hasta el extremo: cuando no se sabe exactamente qué subproblemas resolver, simplemente se re-suelven todos y se almacenan las respuestas para que se puedan utilizar más tarde en la resolución de problemas más grandes. Esta aproximación es de amplia uti-lización en la investigación operativa. Aquí el término «programación» se re-fiere al proceso de formular las restricciones del problema de modo tal que el método sea aplicable. Éste es un arte que no se va a abordar con detalle, excepto para analizar algunos ejemplos. (La «programación» que interesará aquí con-siste en la escritura de programas C++ para encontrar las soluciones.)

Ya se han visto algunos algoritmos que se pueden llevar a los términos de la programación dinámica. Por ejemplo, el algoritmo de Warshall para encontrar la clausura transitiva de un grafo y el algoritmo de Floy para encontrar el ca-mino más corto en un grafo ponderado (ambos en el Capítulo 32) trabajan los dos considerando los vértices uno por uno y resolviendo lo subproblemas del vértice actual haciendo uso de las soluciones de los vértices ya considerados.

Pueden surgir dos dificultades en la aplicación de la programación diná-mica. Primero, puede que no siempre sea posible combinar las soluciones de dos problemas pequeños para obtener la de uno mayor. Segundo, el número de subproblemas a resolver puede ser inaceptablemente grande. Nadie ha llegado a caracterizar con precisión qué problemas se pueden resolver con la progra-mación dinámica; hay muchos problemas «difíciles» para los que este método no parece ser aplicable (ver Capítulos 44 y 45), al igual que muchos problemas «fáciles» para los que es menos eficaz que los algoritmos estándar.

En este capítulo se verán algunos ejemplos de problemas para los que la pro-gramación dinámica es bastante eficaz. Su objetivo es buscar la «mejor» forma de hacer algo y tiene la propiedad de que cualquier decisión que suponga en-

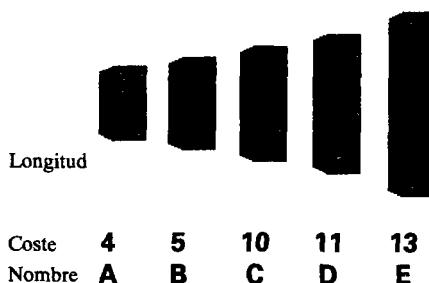


Figura 42.1 Elementos del problema de la mochila.

contrar la mejor forma de resolver un pequeño subproblema es también válida cuando el subproblema pasa a ser una parte de un problema mayor.

El problema de la mochila

Un ladrón que roba una caja fuerte encuentra que está llena con N tipos de objetos de tamaño y valor diferentes, pero sólo tiene una mochila de capacidad M para cargar con el botín. El *problema de la mochila* consiste en encontrar la combinación de objetos que el ladrón debe seleccionar para maximizar el valor total de los elementos que se lleva.

Por ejemplo, supóngase que la mochila tiene capacidad 17 y que la caja fuerte contiene muchos objetos con los tamaños y valores que se muestran en la Figura 42.1. (Como de costumbre, se utilizará una sola letra para los nombres de los objetos del ejemplo e índices enteros en los programas, a sabiendas de que los nombres más complicados se pueden convertir en enteros utilizando las técnicas estándar de búsqueda.) Entonces el ladrón puede tomar cinco A (pero no seis) de un total de 20, o puede llenar la mochila con una D y una E de un total de 24, o pudiera intentar otras muchas combinaciones. ¿Pero cuál maximiza su botín?

Por supuesto, hay muchas situaciones comerciales en las que la solución al problema de la mochila podría ser importante. Por ejemplo, una compañía de transporte quizás quiera saber cuál es la mejor forma de cargar un camión o un avión de carga con los objetos a embarcar. En tales aplicaciones, pueden aparecer también otras variantes del problema: por ejemplo, hay una cantidad limitada disponible de cada objeto. Muchas de estas variantes se pueden tratar con el mismo método que el que se examinará para resolver el problema planteado anteriormente.

En una solución de programación dinámica al problema de la mochila, se calcula la mejor combinación para todos los tamaños de mochila hasta M . Este

cálculo se puede hacer eficazmente organizando los cálculos en el orden apropiado, como en el programa siguiente:

```

for (j = 1; j <= N; j++)
{
    for (i = 1; i <= M; i++)
        if (i >= longitud[j])
            if (coste[i] < coste[i-longitud[j]]+valor[j])
            {
                coste[i] = coste[i-longitud[j]]+valor[j];
                mejor[i] = j;
            }
}

```

En este programa, `coste[i]` es el mayor valor que se puede obtener con una mochila de capacidad i y `mejor[i]` es el último objeto que se añadió para alcanzar ese máximo (como se describirá más adelante, permite recuperar el contenido de la mochila). Primero se calcula lo mejor que se puede hacer para todos los tamaños de la mochila cuando sólo se toman objetos de tipo A; luego se calcula lo mejor que se puede hacer cuando se cogen sólo A y B, etc. La solución se reduce a un simple cálculo de `coste[i]`. Supóngase que un objeto j se elige para poner en la mochila: entonces el mejor valor que se puede alcanzar para el total sería `valor[j]` (para el objeto) más `coste[i-longitud[j]]` (para llenar el resto de la mochila). Si el valor supera al mejor valor que se puede alcanzar sin un objeto j , entonces se actualizan `coste[i]` y `mejor[i]`; en caso contrario se dejan como estaban. Una simple inducción prueba que esta estrategia resuelve el problema.

La Figura 42.2 muestra los cálculos del ejemplo. El primer par de líneas muestra la mejor solución (los contenidos de los arrays `coste` y `mejor`) con sólo A, el segundo con A y B, etc. El mayor valor que se puede obtener con una mochila de tamaño 17 es 24. Durante el cálculo de este resultado, se resuelven muchos subproblemas más pequeños. Por ejemplo, el mayor valor que se puede obtener con una mochila de tamaño 16 utilizando sólo A, B y C es 22.

El contenido óptimo de la mochila se puede calcular con ayuda del array `mejor`. Por definición `mejor[M]` está incluido en la mochila y el resto de contenidos son los mismos que para la mochila óptima de tamaño $M-\text{longitud}[\text{mejor}[M]]$. Por lo tanto, también se incluye `mejor[M-longitud[mejor[M]]]`, y así sucesivamente. Para el ejemplo, `mejor[17] = C`; entonces se encuentra otro objeto de tipo C de tamaño 10, y finalmente un objeto de tipo A de tamaño 3.

Propiedad 42.1 *La solución por programación dinámica del problema de la mochila lleva un tiempo proporcional a NM .*

	k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1	coste[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	mejor[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2	coste[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	mejor[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3	coste[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	mejor[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4	coste[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	mejor[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5	coste[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	mejor[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Figura 42.2 Solución al problema de la mochila.

Esto es evidente a partir del análisis del código.■

De este modo, el problema de la mochila se resuelve fácilmente si M no es muy grande, pero el tiempo de ejecución se hace inaceptable para grandes capacidades. Además, un punto crucial que no puede pasarse por alto es que el método no funciona si M y los tamaños o los valores son, por ejemplo, números reales en lugar de enteros. Esto más que una simple molestia es una dificultad fundamental. No se conoce ninguna buena solución para este problema y verá en el Capítulo 45 que mucha gente cree que no existe dicha solución. Para apreciar este problema, el lector puede tratar de resolver el caso en el que los valores son todos 1, el tamaño del j -ésimo objeto es \sqrt{j} y M es $N/2$. Pero cuando las capacidades, los tamaños y los valores son todos enteros, se tiene el principio fundamental de que las decisiones óptimas, una vez que se toman, no es necesario cambiarlas. Una vez conocida la mejor forma de llenar una mochila de cualquier tamaño con los primeros j objetos, no es necesario reconsiderar esos problemas, independientemente de cuáles sean los nuevos objetos. En cualquier momento que se pueda aplicar este principio general, la programación dinámica es utilizable. Esto es válido en este caso porque los valores enteros permiten tomar decisiones exactas, «óptimas».

En este algoritmo sólo se necesita memorizar una pequeña cantidad de información relativa a las decisiones óptimas anteriores, si M no es muy grande. Las diferentes aplicaciones de la programación dinámica tienen diferentes necesidades a este respecto; a continuación se verán otros ejemplos.

Producto de matrices en cadena

Un clásico problema de programación dinámica consiste en minimizar la cantidad de cálculos que se necesitan para multiplicar una serie de matrices de diferentes tamaños. Tales métodos se deben considerar en cualquier aplicación que implique manipulaciones sistemáticas de matrices.

Supóngase que las seis matrices

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} (d_{11} \ d_{12}) \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

se tienen que multiplicar entre sí. Por supuesto, para que la multiplicación sea válida, el número de columnas de una matriz debe ser el mismo que el número de filas de la siguiente. Pero el número total de multiplicaciones escalares implicadas depende del orden en el que se multipliquen las matrices. Por ejemplo, se puede proceder de izquierda a derecha: multiplicando A por B, se obtiene una matriz de $4*3$ después de 24 multiplicaciones escalares. Multiplicando este resultado por C, se tiene una matriz de $4*1$ después de 12 multiplicaciones escalares más. Al multiplicar este resultado por D, se tiene una matriz de $4*2$ después de 8 multiplicaciones escalares más. Continuando de esta manera, se tiene como resultado una matriz de $4*3$ después de un total de 84 multiplicaciones escalares. Pero si en lugar de esto se procede de izquierda a derecha se obtiene el mismo resultado de $4*3$ con sólo 69 multiplicaciones.

Evidentemente son posibles muchos otros órdenes. El orden de la multiplicación se puede expresar por medio de paréntesis: por ejemplo el orden de izquierda a derecha es el orden (((((AB)C)D)E)F), y de derecha a izquierda es (A(B(C(D(EF))))). Cualquier sistema de paréntesis válido conduce a la respuesta correcta, pero ¿cuál se acompaña de la de menor cantidad de multiplicaciones escalares?

Se pueden alcanzar ahorros sustanciales cuando se trata de grandes matrices: por ejemplo, si las matrices B, C y F del ejemplo anterior fueran todas de dimensión 300 en lugar de 3, entonces el orden de izquierda a derecha necesita 6.024 multiplicaciones escalares pero el de derecha a izquierda utilizaría un astronómico 274.200. (En estos cálculos se supone que se utiliza el método estándar de multiplicación de matrices. El método de Strassen u otro similar podrían en principio ahorrar mucho trabajo con matrices muy grandes, pero se pueden aplicar las mismas consideraciones sobre el orden de las multiplicaciones. Así, al multiplicar una matriz $p*q$ por una $q*r$ se obtendrá una matriz $p*r$, calculando cada entrada con q multiplicaciones, y un total de pqr multiplicaciones.)

En general, supóngase que se multiplican entre sí N matrices:

$$M_1 M_2 M_3 \dots M_N$$

tales que satisfacen la restricción de que M_i tiene r_i filas y r_{i+1} columnas, para $1 \leq i < N$. Se desea encontrar el orden de multiplicación de las matrices que minimice el número total de multiplicaciones a utilizar. Ciertamente tratar todos los órdenes posibles es impráctico. (El número de ordenaciones es una función combinatoria muy bien conocida denominada el *número catalán*: el número de formas de escribir los paréntesis de N variables es alrededor de $4^{N-1}/N\sqrt{(\pi N)}$.) Pero ciertamente merece la pena dedicar cierto esfuerzo a encontrar una buena solución porque N es, en general, bastante pequeño con relación al número de multiplicaciones a realizar.

Como ocurrió antes, la solución por programación dinámica a este problema implica trabajar de forma ascendente memorizando los resultados de los subproblemas parciales para evitar reprocesamiento. En primer lugar, sólo hay una forma de multiplicar M_1 por M_2 , M_2 por M_3 , ..., M_{N-1} por M_N ; guardar los costes. Después se calcula la mejor forma de multiplicar triples sucesivos, utilizando todos los cálculos hechos hasta ahora. Por ejemplo, para encontrar la mejor forma de multiplicar $M_1 M_2 M_3$, primero se encuentra el coste de multiplicar $M_1 M_2$ en la tabla que se memoriza y luego se añade el coste de multiplicar el resultado por M_3 . Este total se compara con el coste de multiplicar primero $M_2 M_3$, y luego multiplicar por M_1 , lo que se puede calcular de la misma forma. El menor de éstos se memoriza, y se sigue el mismo procedimiento para todos los triples. A continuación se calcula la mejor forma de multiplicar grupos sucesivos de cuatro, utilizando la información obtenida hasta ahora. Continuando de esta manera se encuentra la mejor forma de multiplicar entre sí todas las matrices.

Esto conduce al programa siguiente:

```

for (i = 1; i <= N; i++)
    for (j = i+1; j <= N; j++) coste[i][j] = INT_MAX;
for (i = 1; i <= N; i++) coste[i][i] = 0;
for (j = 1; j < N; j++)
    for (i = 1; i <= N-j; i++)
        for (k = i+1; k <= i+j; k++)
        {
            t = coste[i][k-1] + coste[k][i+j]
                r[i]*r[k]*r[i+j+1];
            if (t < coste[i][i+j])
                { coste[i][i+j] = t; mejor[i][i+j] = k; }
        }
    }
}

```

Para $1 \leq j \leq N-1$, se encuentra el coste mínimo de calcular

$$M_i M_{i+1} \dots M_{i+j}$$

buscando, para $1 < i \leq N-j$ y para cada k entre i y $i+j$, el coste de calcular

$M_i M_{i+1} \dots M_{k-1}$ y $M_k M_{k+1} \dots M_{i+j}$ y añadiendo el coste de multiplicar estos resultados entre sí. Puesto que siempre se divide un grupo en otros más pequeños, el coste mínimo para los dos subgrupos se busca simplemente en la tabla, sin tener que recalcularlo. En particular, $\text{coste}[1][r]$ da el coste mínimo de calcular $M_1 M_{l+1} \dots M_r$, $\text{coste}[i][k-1]$ el del primer grupo anterior, y $\text{coste}[k, i+j]$ el del segundo grupo. El coste de la multiplicación final se determina fácilmente: $M_i M_{i+1} \dots M_{k-1}$ es una matriz de $r_i * r_k$, y $M_k M_{k+1} \dots M_{i+j}$ es una matriz $r_k * r_{i+j+1}$, por tanto el coste de multiplicar estas dos es $r_i r_k r_{i+j+1}$. De esta forma, el programa calcula $\text{coste}[i][i+j]$ para $1 \leq i \leq N-j$ con j desde 1 a $N-1$. Cuando se alcanza $j = N - 1$ (e $i = 1$), se ha encontrado el coste mínimo para calcular $M_1 M_2 \dots M_N$ que es lo que se deseaba.

Como antes, se necesita memorizar las decisiones tomadas en un array mejor para su posterior recuperación cuando se generen las series reales de multiplicaciones. El programa siguiente implementa este proceso para extraer el sistema de paréntesis óptimo a partir de los arrays `coste` y `mejor` calculados por el programa anterior:

```
orden(int i, int j)
{
    if (i == j) cout >> nombre(i); else
    {
        cout >> '(';
        orden(i, mejor[i][j]-1); orden(mejor[i][j], j);
        cout >> ')';
    }
}
```

La Figura 42.3 es una tabla que puede ser utilizada para seguir la evolución de estos programas para el ejemplo del problema anterior. En ella se muestra el coste total y la «última» multiplicación óptima para cada serie de la lista de matrices. Por ejemplo, la entrada de la fila A y la columna F indica que se necesitan 36 multiplicaciones escalares para multiplicar las matrices de A a F, y que se puede obtener multiplicando de A a C de forma óptima, luego D a F y multiplicando finalmente las matrices resultantes. Sólo D está realmente en el array `mejor`: las divisiones óptimas se indican en el array para clarificar. Para encontrar cómo multiplicar de A a C de forma óptima, se mira en la fila A y la columna C, etc. En el ejemplo el sistema de paréntesis producido por el programa es $((A(BC))(DE)F)$ que, como se mencionó anteriormente, necesita sólo 36 multiplicaciones escalares. Para el ejemplo citado al principio donde las dimensiones de B, C y F se cambian de 3 a 300, el mismo sistema es óptimo, para lo que se requieren 2.412 multiplicaciones escalares.

	B	C	D	E	F
A	24 [A][B]	14 [A][BC]	22 [ABC][D]	26 [ABC][DE]	36 [ABC][DEF]
B		6 [B][C]	10 [BC][D]	14 [BC][DE]	22 [BC][DEF]
C			6 [C][D]	10 [C][DE]	19 [C][DEF]
D				4 [D][E]	10 [DE][F]
E					12 [E][F]

Figura 42.3 Solución al problema de matrices en cadena.

Propiedad 42.2 *La programación dinámica resuelve el problema del producto de matrices en cadena en un tiempo proporcional a N^3 y en un espacio proporcional a N^2 .*

De nuevo, esto se obtiene directamente del examen del programa. En particular, las necesidades de espacio son sustancialmente mayores que las utilizadas en el problema de la mochila. Pero el tiempo y el espacio necesarios para encontrar el óptimo son posiblemente bastante insignificantes comparados con los ahorros obtenidos.■

Árboles binarios de búsqueda óptima

En muchas aplicaciones de búsqueda se sabe que las claves pueden aparecer con una gran variación de frecuencia. Por ejemplo, un programa que verifique la ortografía de un texto en español posiblemente busque palabras como «y» y «da» mucho más a menudo que palabras como «dinámica» y «programación». De forma similar, un compilador de C++ posiblemente tenga que buscar palabras como «if» y «for» mucho más a menudo que «goto» o «main». Si se utiliza un árbol binario de búsqueda, sería una gran ventaja tener las palabras más frecuentes cerca de la raíz del árbol. Se puede utilizar un algoritmo de programación dinámica para determinar cómo organizar las claves del árbol con el fin de minimizar el coste total de la búsqueda.

Cada nodo del árbol binario de la Figura 42.4 está etiquetado con un entero que se supone proporcional a su frecuencia de acceso. Esto es, de cada 18 búsquedas en el árbol, se espera que cuatro sean de A, dos de B, una de C, etc. Cada

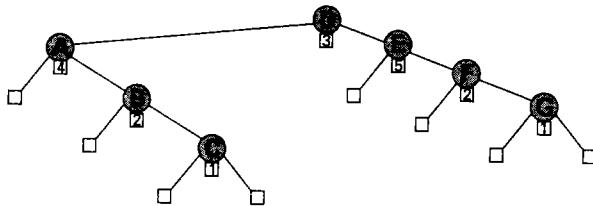


Figura 42.4 Árbol binario de búsqueda con sus frecuencias.

una de las cuatro búsquedas de A necesita dos accesos a nodo, cada una de las dos de B tres y así sucesivamente. Se puede calcular una medida del «coste» del árbol simplemente multiplicando la frecuencia de cada nodo por su distancia a la raíz. Ésta es la *longitud ponderada del camino interno* del árbol. Para el de la Figura 42.4 la longitud ponderada del camino interno es $4 * 2 + 2 * 3 + 1 * 1 + 3 * 3 + 5 * 4 + 2 * 2 + 1 * 3 = 51$. Lo que se quisiera sería encontrar un árbol binario de búsqueda para las mismas claves y las frecuencias de modo que tenga la menor longitud del camino interno de todos los árboles.

Este problema es similar al de minimizar la longitud ponderada del camino externo que se vio al estudiar la codificación de Huffman (Capítulo 22). En ella, sin embargo, no fue necesario mantener el orden de las claves; en el árbol binario de búsqueda se debe preservar la propiedad de que todos los nodos a la izquierda de la raíz tengan claves inferiores, etc. Estos requisitos hacen el problema muy similar al del producto de matrices en cadena anterior: virtualmente se puede utilizar el mismo programa.

Más concretamente, supóngase que se tiene un conjunto de claves de búsqueda $K_1 < K_2 < \dots < K_N$ y las frecuencias asociadas r_0, r_1, \dots, r_N donde r_i es la frecuencia «esperada» de acceso a la clave K_i . Se desea encontrar el árbol binario de búsqueda que minimiza la suma, sobre todas las claves, del producto de sus frecuencias por la distancia a la raíz (el coste de tener acceso al nodo asociado).

La aproximación por programación dinámica de este problema consiste en calcular, para cada j entre 1 y $N - 1$, la mejor manera de construir un subárbol que contenga $K_i, K_{i+1}, \dots, K_{i+j}$ para $1 \leq i \leq N-j$, como en el programa siguiente:

```

for (i = 1; i <= N; i++)
    for (j = i+1; j <= N+1; j++) coste[i][j] = INT_MAX;
for (i = 1; i <= N; i++) coste[i][i] = f[i];
for (i = 1; i <= N+1; i++) coste[i][i-1] = 0;
for (j = 1; j <= N-1; j++)
    for(i = 1; i <= N-j; i++)
    {
        
```

```

for (k = i; k <= i+j; k++)
{
    t = coste[i][k-1] + coste[k+1][i+j];
    if (t < coste[i][i+j])
        { coste[i][i+j] = t; mejor[i][i+j] = k; }
}
for (k = i; k <= i+j; coste[i][i+j] += f[k++]);
}

```

Para cada j , el cálculo se hace tratando a cada nodo como si fuera la raíz y utilizando los valores ya calculados para determinar la mejor forma de construir los subárboles. Para cada k entre i e $i+j$, se desea encontrar el árbol óptimo que contiene $K_i, K_{i+1}, \dots, K_{i+j}$ con K_k como raíz. Este árbol se forma utilizando el árbol óptimo para $K_i, K_{i+1}, \dots, K_{k-1}$ como subárbol izquierdo y el árbol óptimo para $K_{k+1}, K_{k+2}, \dots, K_{i+j}$ como subárbol derecho. La longitud del camino interno de este árbol es la suma de las longitudes del camino interno de los dos subárboles más la suma de las frecuencias de todos los nodos (puesto que cada nodo del nuevo árbol está a un nivel más bajo con respecto a la raíz).

Obsérvese que la suma de todas las frecuencias se añade a cualquier nuevo coste, y por tanto éste no es necesario cuando se busca el mínimo. También se debe tener $\text{coste}[i][i-1]=0$ para cubrir la posibilidad de que un nodo pueda tener solamente un hijo (este caso no tiene un equivalente en el problema de multiplicación de matrices en cadena).

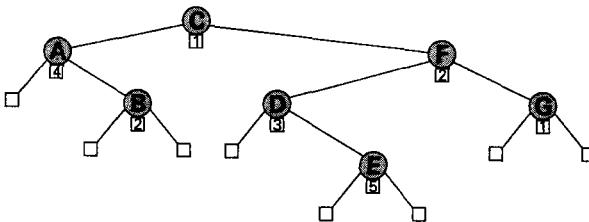


Figura 42.5 Árbol binario de búsqueda óptimo.

Como antes, se necesita un corto programa recursivo para recuperar el árbol real construido por el programa a partir del array `mejor`. El árbol óptimo calculado a partir del de la Figura 42.4 se muestra en la Figura 42.5. La longitud ponderada del camino interno de este árbol es 41.

Propiedad 42.3 *El método de programación dinámica para construir un árbol binario de búsqueda óptimo requiere un tiempo proporcional a N^3 y un espacio proporcional a N^2 .*

Una vez más, el algoritmo necesita una matriz de tamaño N^2 y un tiempo proporcional a N en cada entrada. De hecho, en este caso es posible reducir el tiempo necesario a N^2 explotando el hecho de que la posición óptima de la raíz de un árbol no puede estar muy lejos de la posición óptima de la raíz de un árbol ligeramente más pequeño, por lo que k no necesita recorrer todos los valores de i hasta $i+j$ en el programa anterior. ■

Necesidades de espacio y tiempo

Los ejemplos anteriores sugieren que las aplicaciones de la programación dinámica pueden tener diferentes requisitos de tiempo y espacio dependiendo de la cantidad de información memorizada sobre los subproblemas más pequeños. Para el algoritmo del camino más corto, no se necesita espacio extra; para el problema de la mochila se necesita una memoria proporcional al tamaño de ella; para otros problemas se necesita un espacio proporcional a N^2 . Para cada problema, el tiempo requerido es un factor N veces más grande que la memoria necesaria.

El espectro de posibles aplicaciones de la programación dinámica es mucho mayor que el que se muestra en estos ejemplos. Desde el punto de vista de la programación dinámica, la recursión divide y vencerás debe concebirse como un caso particular en el que se debe calcular y memorizar una cantidad mínima de información sobre los casos más pequeños, y la búsqueda exhaustiva (que se examinará en el Capítulo 44) se puede concebir como un caso particular en el que se debe calcular y memorizar una cantidad máxima de información sobre los casos más pequeños. La programación dinámica es una técnica de diseño natural que se manifiesta de muchas formas entre estos dos extremos.

Ejercicios

1. En el ejemplo dado para el problema de la mochila, los objetos se ordenan por tamaño. ¿Sería válido el algoritmo si aparecieran en un orden arbitrario?
2. Modificar el programa de la mochila para tener en cuenta otra restricción definida por un array que contiene el número de objetos disponibles de cada tipo.
3. ¿Qué haría el programa de la mochila si uno de los valores fuera negativo?
4. Verdadero o falso: Si el producto de matrices en cadena implica una multiplicación $1 * k$ por $k * 1$, existe una solución óptima para la que dicha multiplicación es la última. Explicar la respuesta.
5. Escribir un programa para encontrar la *segunda mejor* forma de multiplicar entre sí una cadena de matrices.

6. Dibujar el árbol binario de búsqueda óptimo para el ejemplo del texto, pero con todas las frecuencias incrementadas en una unidad.
7. Escribir el programa de construcción del árbol binario de búsqueda óptimo.
8. Supóngase que se ha calculado el árbol binario de búsqueda óptimo para un cierto conjunto de claves y frecuencias, y que una de las frecuencias se ha incrementado en una unidad. Escribir un programa para construir el nuevo árbol óptimo.
9. ¿Por qué no se puede resolver el problema de la mochila de la misma forma que el de la multiplicación de matrices en cadena y el del árbol binario de búsqueda óptimo: es decir, minimizando, para k de 1 a M , la suma de los mejores valores posibles para una mochila de tamaño k y una de tamaño $M-k$?
10. Ampliar el programa del problema de los caminos más cortos para incluir un procedimiento `caminos(int i, int j)` que llene un array `camino` con el camino más corto entre i y j . Este procedimiento debería tomar un tiempo proporcional a la longitud del camino a cada llamada, utilizando una estructura de datos auxiliar construida por una versión modificada del programa dado en el Capítulo 32.

Programación lineal

En muchos problemas prácticos aparecen complicadas interacciones entre un cierto número de cantidades variables. Un ejemplo de esto es el problema del flujo de red presentado en el Capítulo 33: los flujos en cada tubería deben obedecer a ciertas leyes físicas de la red. Otro ejemplo es la planificación de tareas (por ejemplo) en un proceso de producción ante fechas de entrega, prioridades, etc. Muy a menudo es posible desarrollar una formulación matemática precisa que capte las interacciones en juego y que reduzca el problema particular a un problema matemático más directo. Este proceso de transformar a un conjunto de ecuaciones cuya solución implica la de un problema práctico dado se denomina *programación matemática*. De nuevo, como en el Capítulo 42, el término «programación» se refiere al proceso de búsqueda de variables y de establecer el sistema de ecuaciones cuya solución corresponda a la del problema. En este capítulo se considerará un tipo fundamental de programación matemática, *la programación lineal*, y un algoritmo eficaz para resolver los programas lineales, el método *símplex*.

La programación lineal y el método simplex son de una importancia fundamental, ya que una gran variedad de problemas importantes se pueden formular como programas lineales y su solución se alcanza eficazmente a través del método simplex. Se conocen mejores algoritmos para algunos problemas específicos, pero muy pocas técnicas de resolución de problemas tienen un espectro de aplicación tan grande como el proceso de formular primero el problema como un programa lineal y calcular la solución aplicando el método simplex. Una rutina de biblioteca para el método simplex puede ser una herramienta indispensable para resolver problemas complejos.

Las investigaciones en programación lineal son muy extensas y una perfecta comprensión de todos los factores implicados requiere una madurez matemática más allá de la que se supone en este libro. Por otra parte, algunas de las ideas básicas son fáciles de comprender y el actual algoritmo simplex no es difícil de implementar, como se verá más adelante. Como con la transformada rápida de Fourier del Capítulo 41, la intención no es dar una implementación

práctica completa, sino más bien aprender algunas de las propiedades básicas del algoritmo y sus relaciones con otros algoritmos que se han estudiado ya.

Programas lineales

Los programas matemáticos implican a un conjunto de *variables* relacionadas por un conjunto de ecuaciones matemáticas (*restricciones*) y una *función objetivo* que contiene a las variables y que debe maximizarse respetando las restricciones dadas. Si todas las ecuaciones en juego son simples combinaciones lineales de las variables, se tiene el caso particular que se está considerando y que se denomina *programación lineal*.

El siguiente programa lineal corresponde al problema del flujo de red del Capítulo 33.

*Maximizar $x_{AB} + x_{AD}$
respetando las restricciones*

$$\begin{array}{ll} x_{AB} \leqslant 6 & x_{CD} \leqslant 3 \\ x_{AC} \leqslant 8 & x_{CE} \leqslant 3 \\ x_{BD} \leqslant 6 & x_{DF} \leqslant 8 \\ x_{BE} \leqslant 3 & x_{EF} \leqslant 6 \end{array}$$

$$x_{BD} + x_{BE} = x_{AB},$$

$$x_{CD} + x_{CE} = x_{AC},$$

$$x_{BD} + x_{CD} = x_{DF},$$

$$x_{BE} + x_{CE} = x_{EF},$$

$$x_{AB}, x_{AC}, x_{BD}, x_{BE}, x_{CD}, x_{CE}, x_{DF}, x_{EF} \geqslant 0.$$

Cada variable de este programa lineal corresponde al flujo de cada una de las tuberías. Estas variables satisfacen dos tipos de ecuaciones: desigualdades, que corresponden a las restricciones de capacidad de las tuberías, e igualdades, que corresponden a las restricciones del flujo en cada unión. De modo que, por ejemplo, la desigualdad $x_{AB} \leqslant 8$ significa que la tubería AB tiene una capacidad de 8, y la ecuación $x_{BD} + x_{BE} = x_{AB}$ indica que en la unión B el flujo de salida debe ser igual al flujo de entrada. Se observa que todas las igualdades juntas dan la restricción implícita $x_{AB} + x_{AC} = x_{DF} + x_{EF}$, que dice que todo el flujo de entrada en la red debe ser igual al flujo de salida en toda la red. También, por supuesto, todos los flujos deben ser positivos.

Ésta es evidentemente una formulación matemática del problema de flujo

de red: una solución a este problema matemático particular es una solución al caso particular del problema de flujo de red. Este ejemplo no se ha elegido para demostrar que la programación lineal proporcionará un algoritmo mejor para este problema particular, sino que es una técnica tan general que se puede aplicar en un gran número de problemas. Por ejemplo, si se fuera a generalizar el problema de flujo de red incluyendo costes, por ejemplo, además de las capacidades, la formulación de la programación lineal no sería muy diferente, aun cuando el problema pudiera ser significativamente más difícil de resolver directamente.

Los problemas lineales no sólo son muy explícitos, sino que también existe un algoritmo para resolverlos (el algoritmo simplex) que ha demostrado ser muy eficaz en muchos problemas que se presentan en la práctica. Para algunos problemas (tales como el de flujo de red) existen algoritmos específicamente orientados al problema con mejor rendimiento que la programación lineal/simplex; para otros (incluyendo varias extensiones de flujo de red), no se conocen algoritmos mejores. Aunque existe un algoritmo mejor, puede ser complicado o difícil de implementar, mientras que el procedimiento de desarrollo y resolución de un programa lineal con la rutina de biblioteca simplex es el más inmediato. Este aspecto «generalista» del método es bastante atractivo y explica su amplia utilización. El peligro de confiar demasiado en él puede llevar a soluciones ineficaces para algunos problemas simples (por ejemplo, muchos de los estudiados en este libro).

Interpretación geométrica

Los programas lineales pueden tener una representación geométrica. El siguiente es fácil de visualizar, ya que sólo implica dos variables:

*Maximizar $x_1 + x_2$
respetando las restricciones*

$$-x_1 + x_2 \leq 5,$$

$$x_1 + 4x_2 \leq 45,$$

$$2x_1 + x_2 \leq 27,$$

$$3x_1 - 4x_2 \leq 24,$$

$$x_1, x_2 \geq 0.$$

Este programa lineal corresponde a la situación geométrica dibujada en la Figura 43.1. Cada desigualdad define un semiplano en el que debe encontrarse toda solución al programa lineal. Por ejemplo, $x_1 \geq 0$ significa que toda solu-

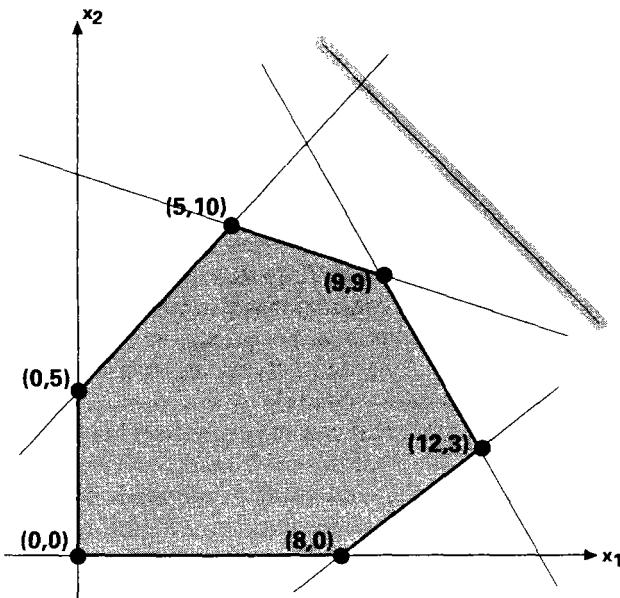


Figura 43.1 Un simplex bidimensional.

ción debe estar a la derecha del eje x_2 , y $-x_1 + x_2 < 5$ significa que toda solución debe estar por debajo y a la derecha de la línea $-x_1 + x_2 = 5$ (la que pasa por $(0,5)$ y $(5,10)$). Toda solución al programa lineal debe satisfacer *todas* estas restricciones, por lo que la región definida por la intersección de todos estos semiplanos (sombreada en la figura) es el conjunto de todas las soluciones posibles. Para resolver el programa lineal se debe encontrar el punto interior de esta región que maximiza la función objetivo.

Una región definida por intersecciones de semiplanos es siempre un conjunto convexo (ya se ha encontrado este hecho antes, en una de las definiciones del cerco convexo del Capítulo 25). Esta región convexa, denominada el *símplex*, forma la base de un algoritmo para encontrar la solución al programa lineal que maximiza la función objetivo.

Una propiedad fundamental del simplex explotada por el algoritmo es que la función objetivo es máxima en uno de los vértices del simplex; de modo que sólo es necesario examinar los vértices, y no todos los puntos internos. Para comprender esto, se considera la línea sombreada, arriba a la derecha de la Figura 43.1, que corresponde a la función objetivo. Esta función puede interpretarse como una recta de pendiente conocida (en este caso -1) y posición desconocida. Se desea conocer el punto en el que la recta toca al simplex al moverse

desde el infinito. Este punto es la solución del programa lineal: satisface todas las desigualdades, ya que está en el simplex, y maximiza la función objetivo porque no se pueden encontrar puntos con valores mayores. Para el ejemplo, la recta toca al simplex en el punto (9, 9) en el que la función objetivo toma el valor máximo 18.

Cada función objetivo corresponde a una recta con otra pendiente, pero el valor máximo siempre se alcanza en uno de los vértices del simplex. El algoritmo que se examinará posteriormente es una forma sistemática de moverse desde un vértice a otro en busca del mínimo. En dos dimensiones no existe mucha más elección a hacer, pero, como se verá, el simplex es un objeto mucho más complicado cuando se trata con más variables.

Se puede también apreciar en la Figura 43.1 por qué los programas matemáticos que tratan con funciones no lineales son mucho más difíciles de manipular. Por ejemplo, si la función objetivo no es lineal, puede ser una curva que podría tocar al simplex en uno de sus lados y no en un vértice. Si las desigualdades son también no lineales, se podrían encontrar formas geométricas bastante más complicadas.

La intuición geométrica demuestra claramente que pueden generarse situaciones anómalas. Por ejemplo, supóngase que se añade la desigualdad $x_1 \geq 13$ al programa lineal del ejemplo anterior. Está bastante claro en la Figura 43.1 que en este caso la intersección de los semiplanos es vacía. Tal programa lineal se denomina *no factible*: no existen puntos que satisfagan las desigualdades y por lo tanto ninguno maximiza la función objetivo. Al contrario, la desigualdad $x_1 \leq 13$ es *redundante*: el simplex está totalmente contenido dentro de este semiplano, de tal manera que la desigualdad no está representada en él. Las desigualdades redundantes no afectan para nada a la solución, pero deben eliminarse durante la búsqueda de la solución.

Un problema más serio es que el simplex puede ser una región abierta (no acotada), en cuyo caso la solución puede no estar bien definida. Este podría ser el caso del ejemplo si se suprimieran la segunda y tercera desigualdades. Aun si el simplex no está acotado, la solución puede estar bien definida para algunas funciones objetivos, pero un algoritmo debe encontrar grandes dificultades para moverse en la región no acotada.

Es preciso insistir que, aunque estos problemas son bastante fáciles de ver cuando se trata de dos variables y algunas pocas desigualdades, son mucho más complejos de lo que aparentan en un problema más general con muchas variables y desigualdades. De hecho, la detección de estas situaciones anómalas es una parte importante de las complicaciones de cálculo al resolver los programas lineales.

La misma intuición geométrica es también útil para más de dos variables. En tres dimensiones el simplex es un sólido convexo tridimensional definido por la intersección de semi-espacios definidos por los planos cuyas ecuaciones se obtienen al cambiar las desigualdades por igualdades. Por ejemplo, si se añaden las desigualdades $x_3 \leq 4$ y $x_3 \geq 0$ al programa lineal anterior, el simplex se convierte en el objeto sólido dibujado en la Figura 43.2.

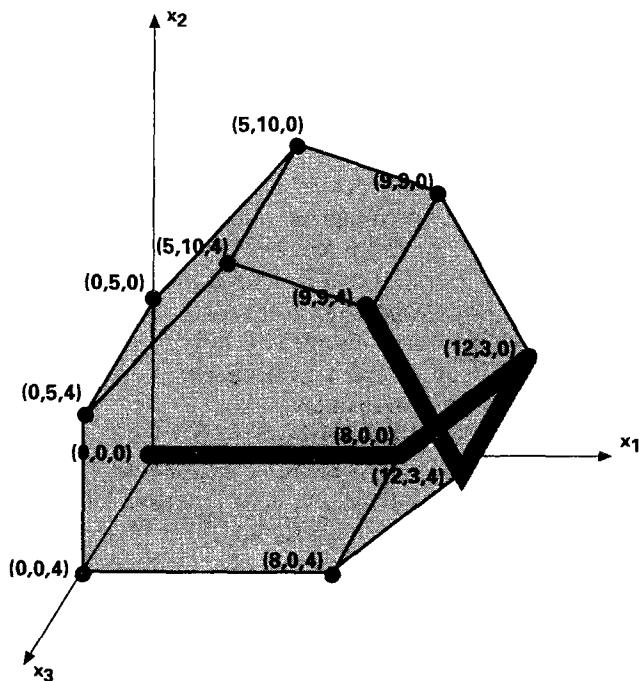


Figura 43.2 Un simpex tridimensional.

Para hacer el ejemplo más tridimensional, supóngase que se cambia la función objetivo por $x_1 + x_2 + x_3$, que define un plano perpendicular a la recta $x_1 = x_2 = x_3$. Si se desplaza el plano desde el infinito a lo largo de esta recta, toca al simplex en el punto $(9, 9, 4)$, que es la solución. (La Figura 43.2 muestra también el camino a través de los vértices del simplex desde $(0, 0, 0)$ a la solución, para referencia en la descripción del algoritmo posterior.)

En n dimensiones, se intersecan semi-espacios definidos por hiperplanos de $(n - 1)$ dimensiones para definir el simplex n dimensional, y desplazar un hiperplano $(n - 1)$ dimensional desde el infinito para intersecar al simplex en el punto solución. Como se mencionó anteriormente, existe el riesgo de una simplificación excesiva al concentrarse en las situaciones intuitivas de dos y tres dimensiones, pero las demostraciones de las propiedades anteriores de convexidad, intersecciones de hiperplanos, etc., necesitan conocimientos de álgebra lineal que están fuera del alcance de este libro. Aun así, la intuición geométrica es muy valiosa, ya que puede ayudar a comprender las características fundamentales del método básico utilizado en la práctica para resolver problemas de mayores dimensiones.

El método simplex

El *método simplex* es el nombre con que se suele describir el enfoque general a la solución de programas lineales utilizando el pivotado, la misma operación fundamental utilizada en la eliminación de Gauss. Esto indica que el pivotado corresponde de una forma natural con la operación geométrica de desplazarse desde un punto del simplex a otro en busca de la solución. Los diversos algoritmos utilizados comúnmente difieren en el detalle esencial del orden en que se buscan los vértices del simplex. Esto es, el muy conocido «algoritmo» para este problema se puede describir con más precisión como un método genérico que puede ser mejorado de diferentes maneras. Se ha encontrado ya este tipo de situación anteriormente, por ejemplo en la eliminación de Gauss (Capítulo 37) o en el algoritmo de Ford-Fulkerson (Capítulo 33).

En principio, está claro que los programas lineales pueden tomar muchas formas diferentes. Por ejemplo, el programa lineal para el problema del flujo de red tiene una mezcla de igualdades y desigualdades, pero los ejemplos geométricos anteriores sólo utilizan desigualdades. Es conveniente reducir el número de posibilidades en alguna medida insistiendo en que todos los programas lineales se presenten bajo la misma *forma estándar*, en la que todas las ecuaciones son igualdades excepto una desigualdad por cada variable para indicar que no es negativa. Esto puede parecer una restricción muy estricta, pero no es difícil convertir un programa lineal general a esta forma estándar. El siguiente programa lineal es la forma estándar del ejemplo tridimensional de la Figura 43.2:

*Maximizar $x_1 + x_2 + x_3$
respetando las restricciones*

$$-x_1 + x_2 + y_1 = 5$$

$$x_1 + x_2 + y_2 = 45$$

$$2x_1 + x_2 + y_3 = 27$$

$$3x_1 - 4x_2 + y_4 = 24$$

$$x_2 + y_5 = 4$$

$$x_1, x_2, x_3, y_1, y_2, y_3, y_4, y_5 \geq 0.$$

Cada desigualdad que implique más de una variable se convierte en una igualdad introduciendo una nueva variable. Las y se denominan variables de *holgura*, ya que compensan la *holgura* permitida por las desigualdades. Cualquier desigualdad que implique una sola variable se puede convertir a la restricción estándar positiva renombrándola simplemente. Por ejemplo, una restricción tal como $x_3 \leq -1$ se trata reemplazando x_3 por $-1 - x'_3$ allí donde aparezca.

Esta formulación hace evidente el paralelismo entre la programación lineal y los sistemas de ecuaciones simultáneas. Si se tienen N ecuaciones con M incógnitas, con la restricción de ser todas positivas, en este caso se observa que existen N variables de holgura, una por cada ecuación (ya que se comienza con N desigualdades). Se supone que $M > N$, lo que implica que existen muchas soluciones a las ecuaciones: el problema consiste en encontrar una que maximice la función objetivo.

Para el ejemplo, existe una solución trivial a las ecuaciones: tomar $x_1 = x_2 = x_3 = 0$, y después asignar valores apropiados a las variables de holgura para satisfacer las igualdades. Esto es válido porque en este ejemplo sucede que $(0, 0, 0)$ es un punto del simplex. Aunque éste no es necesariamente el caso general, en la explicación del método simplex se limitará la atención por ahora a los programas lineales de este tipo, ya que engloban todavía a una clase bastante grande de programas lineales: por ejemplo, si todos los números de la parte derecha de las desigualdades en la forma estándar del programa lineal son positivos y las variables de holgura tienen coeficientes positivos (como en el ejemplo), entonces existe claramente una solución con todas las variables originales cero. Más tarde se volverá al caso general.

Dada una solución con $M - N$ variables iguales a cero, se desprende que se puede encontrar otra solución con la misma propiedad utilizando una operación familiar, *el pivotado*. Se trata esencialmente de la misma operación utilizada en la eliminación de Gauss: se selecciona un elemento a $[p] [q]$ de la matriz de coeficientes definidos por las ecuaciones, después se multiplica la p -ésima fila por un escalar apropiado y se suma a todas las filas restantes para llenar la q -ésima columna de ceros excepto el elemento de la fila q que se pone a 1.

Para ver cómo el pivotado proporciona una forma de resolver los programas lineales, se considera la siguiente matriz, que representa al programa lineal anterior:

$$\left(\begin{array}{ccccccccc} -1,00 & -1,00 & -1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 \\ -1,00 & 1,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 5,00 \\ 1,00 & 4,00 & 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 45,00 \\ 2,00 & 1,00 & 0,00 & 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 27,00 \\ 3,00 & -4,00 & 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & 0,00 & 24,00 \\ 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & 4,00 \end{array} \right)$$

Esta matriz $(N + 1) * (M + 1)$ contiene los coeficientes del programa lineal en su forma estándar; la $(M + 1)$ -ésima columna contiene los valores del lado derecho de las ecuaciones (como en la eliminación de Gauss) y la fila 0 contiene los coeficientes de la función objetivo, con el signo cambiado. El significado de la fila 0 se presenta más adelante; por ahora se tratará con el resto de las filas.

Para el ejemplo, se realizarán todos los cálculos con dos decimales. Al hacer esto evidentemente se ignoran hechos tales como la precisión y el error acumulado, que son tan importantes aquí como en el método de eliminación de Gauss.

Las variables que corresponden a una solución se denominan variables *básicas* y las que se ponen a 0 para encontrar la solución se denominan variables *no básicas*. En la matriz, las columnas correspondientes a las variables básicas tienen exactamente un 1 y el resto de valores 0, mientras que las variables no básicas corresponden a las columnas con más de un valor diferente de 0.

Ahora se supone que se desea elegir como pivote de la matriz el elemento de $p = 4$ y $q = 1$. Es decir, se añade un múltiplo apropiado de la cuarta fila a cada una de las otras filas para hacer toda la primera columna cero excepto en la fila 4 donde se debe tener un 1. Esto produce el siguiente resultado:

$$\left(\begin{array}{ccccccccc} 0,00 & -2,33 & -1,00 & 0,00 & 0,00 & 0,00 & 0,33 & 0,00 & 8,00 \\ 0,00 & -0,33 & 0,00 & 1,00 & 0,00 & 0,00 & 0,33 & 0,00 & 13,00 \\ 0,00 & 5,33 & 0,00 & 0,00 & 1,00 & 0,00 & -0,33 & 0,00 & 37,00 \\ 0,00 & 3,67 & 0,00 & 0,00 & 0,00 & 1,00 & -0,67 & 0,00 & 11,00 \\ 1,00 & -1,33 & 0,00 & 0,00 & 0,00 & 0,00 & 0,33 & 0,00 & 8,00 \\ 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & 4,00 \end{array} \right)$$

Esta operación extrae a la séptima columna de la base y pone en ella a la primera. Se elimina exactamente una columna básica ya que exactamente una columna básica tiene un 1 en la fila p .

Por definición, se puede obtener una solución a un programa lineal poniendo todas las variables no básicas a 0, y utilizando la solución trivial dada en la base. En la solución correspondiente a la matriz anterior, x_2 y x_3 son los dos cero, ya que son variables no básicas, y $x_1 = 8$, dado que la matriz corresponde al punto $(8, 0, 0)$ del simplex. (No interesan los valores de las variables de holgura.) Se observa que el vértice superior derecho de la matriz (fila 0, columna $M + 1$) contiene el valor de la función objetivo en este punto. Esto es por diseño, como pronto se verá.

Ahora se supone que se realiza la operación de elegir como pivote el elemento de $p = 3$ y $q = 2$:

$$\left(\begin{array}{ccccccccc} 0,00 & 0,00 & -1,00 & 0,00 & 0,00 & 0,64 & -0,09 & 0,00 & 15,00 \\ 0,00 & 0,00 & 0,00 & 1,00 & 0,00 & 0,09 & 0,27 & 0,00 & 14,00 \\ 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & -1,45 & 0,64 & 0,00 & 21,00 \\ 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,27 & -0,18 & 0,00 & 3,00 \\ 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,36 & 0,09 & 0,00 & 12,00 \\ 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & 4,00 \end{array} \right)$$

Esto elimina la columna 6 de la base y añade la columna 2. Poniendo las variables no básicas a 0 y evaluando las variables básicas como se hizo anteriormente, se ve que esta matriz corresponde al punto $(12, 3, 0)$ del simplex, para el que la función objetivo tiene el valor 15. Se observa que el valor de la función objetivo es estrictamente creciente, lo que, de nuevo es por diseño, como pronto se verá.

¿Cómo decidir qué valores de p y q utilizar como pivote? Es aquí donde en-

tra en juego la fila 0. Para cada variable no básica, la fila 0 contiene la cantidad en la que se podría incrementar la función objetivo si dicha variable pasara de 0 a 1, con el signo cambiado. (El signo se cambia de tal forma que la operación de pivotado estándar mantendrá la fila 0 sin cambios.) El pivotado utilizando la columna q cambia el valor de la variable correspondiente de 0 a algún valor positivo, de modo que se puede asegurar que la función objetivo aumentará si se utiliza cualquier columna con un valor negativo en la fila 0.

La elección de un pivote en toda fila que tenga un elemento positivo para esta columna incrementará la función objetivo, pero también hay que garantizar que engendrará una matriz correspondiente a un punto del simplex. Aquí el principal problema es que uno de los elementos de la columna $M + 1$ puede hacerse negativo. Esto se puede evitar hallando, entre los elementos positivos de la columna q (excluyendo la fila 0), el que da el valor más pequeño cuando se divide por el $(M + 1)$ -ésimo elemento de la misma fila. Si se toma p como el índice de la fila que contiene a este elemento y se hace el pivotado, entonces se puede asegurar que la función objetivo se incrementará y que ninguno de los elementos de la columna $M + 1$ se hará negativo; esto es suficiente para asegurar que la matriz resultante corresponde a un punto del simplex.

Existen dos dificultades potenciales con este procedimiento de búsqueda de la fila del pivote. Primero, ¿qué pasa si no existen valores positivos en la q -ésima columna? Ésta es una situación inconsistente: un valor negativo en la fila 0 indica que la función objetivo puede todavía crecer, pero no existe forma de incrementarla. Se demuestra que esta situación se da si y sólo si el simplex no está acotado, de modo que el algoritmo puede terminar e informar el problema. Una dificultad más indirecta aparece en el caso degenerado cuando el $(M + 1)$ -ésima valor de alguna fila (con un valor positivo en la columna q) es 0. Entonces esta fila será seleccionada, pero la función objetivo se incrementará en 0. Esto no es una dificultad por sí misma, pero es un problema cuando existen dos filas como ésa. Ciertas estrategias naturales para la selección de tales filas conducen a un *bucle infinito*: una secuencia infinita de pivotes que no incrementan la función objetivo. Aquí se han estado evitando dificultades (como las del bucle infinito) en el ejemplo para hacer la descripción del método más clara, pero hay que insistir en que tales casos degenerados pueden aparecer a menudo en la práctica. El carácter genérico de la programación lineal implica que los casos degenerados del problema general aparecerán en la solución de problemas específicos.

Para evitar el bucle infinito, existen algunas posibilidades. Un método consiste en romper el bucle aleatoriamente. Esto hace al ciclo improbable en extremo (pero no matemáticamente imposible). Más adelante se descubrirá otra estrategia anti-bucle.

En el ejemplo, se puede escoger el pivote $q = 3$ (porque se tiene un -1 en la fila 0 columna 3) y $p = 5$ (porque 1 es el único valor positivo de la columna 3). Esto proporciona la siguiente matriz:

$$\left(\begin{array}{ccccccccc} 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,64 & -0,09 & 1,00 & 19,00 \\ 0,00 & 0,00 & 0,00 & 1,00 & 0,00 & 0,09 & 0,27 & 0,00 & 14,00 \\ 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & -1,45 & 0,64 & 0,00 & 21,00 \\ 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,27 & -0,18 & 0,00 & 3,00 \\ 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,26 & 0,09 & 0,00 & 12,00 \\ 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & 4,00 \end{array} \right)$$

que corresponde al punto (12, 3, 4) del simplex, por lo que el valor de función objetivo es 19.

En general, podrían existir varios valores negativos en la fila 0, y se han sugerido diversas estrategias diferentes para elegir entre ellas. Hasta aquí se ha estado procediendo de acuerdo con uno de los métodos más populares, el del *mayor incremento*: seleccionar siempre la columna con el valor más pequeño en la fila 0 (la mayor en valor absoluto). Esto no lleva necesariamente al mayor incremento de la función objetivo, ya que la escalación debe hacerse de acuerdo con la fila p seleccionada. Si esta política de selección de columna se combina con la de selección de filas utilizando, en caso de bucles, la fila que resulta de la eliminación de la columna de índice más bajo, entonces no se producen bucles infinitos. (Esta estrategia anti-bucle infinito se debe a R. G. Bland.) Otra posibilidad para la selección de la columna es calcular realmente la cantidad en la que se puede incrementar la función objetivo para cada columna, y después utilizar la que da el crecimiento máximo. Esto se denomina el método de *la mayor pendiente*. Una última posibilidad interesante es seleccionar aleatoriamente entre todas las columnas disponibles.

Por último, después de una elección del pivote $p = 2$ y $q = 7$, se obtiene la solución:

$$\left(\begin{array}{ccccccccc} 0,00 & 0,00 & 0,00 & 0,00 & 0,14 & 0,43 & 0,00 & 1,00 & 22,00 \\ 0,00 & 0,00 & 0,00 & 1,00 & -0,43 & 0,71 & 0,00 & 0,00 & 5,00 \\ 0,00 & 0,00 & 0,00 & 0,00 & 1,57 & -2,29 & 1,00 & 0,00 & 33,00 \\ 0,00 & 1,00 & 0,00 & 0,00 & 0,29 & -0,14 & 0,00 & 0,00 & 9,00 \\ 1,00 & 0,00 & 0,00 & 0,00 & -0,14 & 0,57 & 0,00 & 0,00 & 9,00 \\ 0,00 & 0,00 & 1,00 & 0,00 & 0,00 & 0,00 & 0,00 & 1,00 & 4,00 \end{array} \right)$$

Ésta corresponde al punto (9, 9, 4) del simplex, que maximiza la función objetivo en 22. Todos los valores de la fila 0 son no negativos, de modo que cualquier pivotado sólo servirá para disminuir el valor de la función objetivo.

El ejemplo anterior sintetiza el método simplex para la resolución de programas lineales. En resumen, si se comienza con una matriz cuyos coeficientes corresponden a un punto del simplex, se puede hacer una serie de aplicaciones del pivotado para desplazarse por los puntos adyacentes del simplex, aumentando siempre el valor de la función objetivo, hasta que se alcanza el máximo.

Un factor fundamental, que no se ha destacado todavía, es crucial para el buen funcionamiento de este procedimiento: una vez que se ha alcanzado un punto en el que un simple pivotado no puede mejorar la función objetivo (un

máximo «local»), se ha alcanzado el máximo «global». Ésta es la base del algoritmo simplex. Como se mencionó antes, la prueba de esto (y de muchos otros hechos que parecen evidentes en la interpretación geométrica) está fuera del alcance de este libro. Pero el algoritmo simplex para el caso general opera esencialmente de la misma manera que para el problema explicado aquí.

Implementación

La implementación del método simplex para el caso descrito con anterioridad se deduce directamente de la descripción. En primer lugar, el necesario procedimiento del pivote utiliza un código similar a la implementación de la eliminación de Gauss del Capítulo 37;

```
pivote(int p, int q)
{
    int j, k;
    for (j = 0; j <= M; j++)
        for (k = M+1; k >=1; k--)
            if (j !=p && k!=q)
                a[j][k] = a[j][k]-a[p][k]*a[j][q]/a[p][q];
    for (j = 0; j <= N; j++0
        if (j != p) a[j][q] = 0;
    for (k = 1; k <= M=1; k++)
        if (k != q) a[p][k] = a[p][k]/a[p][q];
    a[p][q] = 1;
}
```

Este programa añade múltiplos de la fila p a cada fila, con el fin de llenar la columna q de ceros, excepto un 1 en la fila q , como ya se dijo. Como en el Capítulo 37, es necesario tener cuidado de no cambiar el valor de $a[p][q]$ antes de que se haya terminado de utilizarlo. En la eliminación de Gauss, se procesaban las filas por debajo de p durante la eliminación ascendente y sólo las filas por encima de p durante la sustitución descendente, utilizando el método de Gauss-Jordan. Un sistema de N ecuaciones lineales con N incógnitas se puede resolver llamando a $\text{pivote}(i,i)$ para i desde 1 hasta N y volviendo nuevamente a 1.

El algoritmo simplex consiste simplemente en hallar los valores de p y q de la forma descrita anteriormente y llamar a pivote , repitiendo el proceso hasta que se alcance el óptimo o se determine que el simplex no está acotado.

```
for (;;)
```

```

{
    for (q = 0; (q<=M+1) && (a[0][q]>=0); q++) ;
    for (p = 0; (p<=N+1) && (a[p][q]<=0); p++) ;
    if (q>M || p>N) break;
    for (i = p+1; i <= N; i++)
        if (a[i][q] > 0)
            if (a[i][M+1]/a[i][q] < a[p][M+1]/a[p][q])
                p = i;
            pivot(p,q);
}

```

Si el algoritmo termina con $q=M+1$ entonces se ha encontrado la solución óptima, el valor alcanzado por la función objetivo está en $a[0][M+1]$ y los valores de las variables se pueden recuperar de la base. Si el programa termina con $p=N+1$, entonces se ha detectado una situación de no acotamiento.

Este programa ignora el problema de cómo evitar el ciclo infinito. Para implementar el método de Bland, es necesario seguir la pista de la columna que habría abandonado la base, si se hubiera hecho el pivotado utilizando la fila p . Esto se hace fácilmente poniendo $\text{fuerab}[p]$ a q después de cada pivotado. Entonces el bucle para calcular p se puede modificar para dar a p el valor i si hay igualdad en las comprobaciones de relación y se cumple que $\text{fuerab}[p] < \text{fuerab}[q]$. Alternativamente, se podría seleccionar un elemento aleatorio generando un entero aleatorio x y reemplazando cada referencia al array $a[p][q]$ (o $a[i][q]$) por $a[(p+x)\%(N+1)][q]$ (o $a[(i+x)\%(N+1)][q]$). Esto tiene el efecto de buscar a través de la columna q de la misma forma anterior, pero comenzando con un punto aleatorio y no por el principio. El mismo tipo de técnica se podría utilizar para seleccionar una columna aleatoria (con un valor negativo en la fila 0) para elegir el pivote.

El programa y el ejemplo anteriores tratan un caso simple que ilustra el principio que hay detrás del algoritmo simplex, pero que evita las complicaciones que aparecen en las aplicaciones reales. La omisión principal es que el programa necesita que la matriz tenga *una base realizable*: un conjunto de filas y columnas que se puedan transformar en la matriz identidad. El programa comienza con la hipótesis de que existe una solución anulando las $M - N$ variables que aparecen en la función objetivo y que la submatriz $N^* N$ que envuelve a las variables de holgura ha sido «resuelta» convirtiéndola en la matriz identidad. Esto es fácil de hacer para el tipo particular de programas lineales que se han establecido (con todas las desigualdades en variables positivas), pero en general se necesita encontrar algún punto del simplex. Una vez que se ha encontrado una solución, se pueden realizar las transformaciones apropiadas (aplicando ese punto al origen) para llevar la matriz a la forma estándar; pero al principio no se sabe siquiera si existe una solución.

En efecto, se ha demostrado que *detectar* si existe una solución es tan difícil como encontrar la solución óptima, si es que existe. De modo que no es sor-

prendente que la técnica más comúnmente utilizada para detectar la existencia de una solución ¡sea el algoritmo simplex! Específicamente, se añade un conjunto de variables artificiales s_1, s_2, \dots, s_N y se introduce la variable s_i en la i -ésima ecuación. Esto se hace simplemente añadiendo a la matriz N columnas con la matriz identidad. Esto da de inmediato una base realizable para este nuevo programa lineal. Entonces se ejecuta el algoritmo anterior con la función objetivo $-s_1 - s_2 - \dots - s_N$. Si existe una solución al programa lineal original, entonces esta función objetivo puede ser maximizada a cero. Si el máximo alcanzado no es cero, entonces el programa lineal original no es realizable. Si el máximo es cero, entonces la situación normal es que s_1, s_2, \dots, s_N son todas variables no básicas, y así se ha calculado una base realizable para el programa lineal original. En los casos degenerados, algunas de las variables artificiales pueden permanecer en la base, y entonces es necesario realizar nuevos pivotados para eliminarlas (sin cambiar el coste).

En resumen, por lo regular se utiliza un proceso de dos fases para resolver programas lineales generales. Primero, se resuelve el programa lineal que contiene a las variables artificiales s , para obtener un punto del simplex para el problema original. Despues, se eliminan las variables s y se introduce la función objetivo original para buscar desde aquí la solución.

El análisis del tiempo de ejecución del método simplex es extremadamente difícil, y se dispone de pocos resultados. Nadie conoce la «mejor» estrategia para la selección del pivote, ya que no existen resultados que digan cuántos pasos pueden esperarse para cualquier clase razonable de problemas. Es posible construir ejemplos artificiales para los que el tiempo de ejecución del simplex es muy grande (una función exponencial del número de variables). Sin embargo, aquellos que han utilizado el algoritmo en la práctica pueden asegurar que es muy eficaz en la resolución de problemas reales.

La versión simplificada del algoritmo simplex que se ha considerado, aunque bastante útil, no es más que una parte general y bella de la rama de las matemáticas que proporciona un conjunto de herramientas a utilizar para resolver una gama de importantes problemas prácticos.

Ejercicios

1. Dibujar el simplex definido por las *desigualdades* $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_1 + 2x_2 \leq 20$, y $x_1 + x_2 + x_3 \leq 10$.
2. Obtener la serie de matrices producida por el ejemplo del texto si la columna de pivote elegido es el mayor q para el que $a[0][q]$ es negativo.
3. Obtener la serie de matrices producida por el ejemplo del texto para la función objetivo $x_1 + 5x_2 + x_3$.
4. Describir lo que sucede cuando se ejecuta el algoritmo simplex sobre una matriz con una columna sólo con ceros.

5. ¿Utiliza el algoritmo simplex el mismo número de pasos si se permutan las filas de la matriz de entrada?
6. Expresar una formulación de programación lineal para el ejemplo del problema de la mochila del Capítulo 42.
7. ¿Cuántos pasos de pivotado se necesitan para resolver el programa lineal «maximizar $x_1 + \dots + x_M$ respetando las restricciones $x_1, \dots, x_M \leq 1$ y $x_1, \dots, x_M \geq 0$ »?
8. Construir un programa lineal que consista en N desigualdades de dos variables para las que el algoritmo simplex necesita al menos $N/2$ pivotes.
9. Expresar un problema de programación lineal en tres dimensiones que ilustre la diferencia entre los métodos del mayor incremento y de la mayor pendiente.
10. Modificar la implementación dada en el texto para escribir las coordenadas del punto de solución óptima.

Búsqueda exhaustiva

Algunos problemas implican el examen de un gran número de soluciones potenciales para encontrar una respuesta y no parecen ser adecuados para su resolución por algoritmos eficaces. En este capítulo se examinarán algunas características de los problemas de este tipo y ciertas técnicas que han demostrado su utilidad para resolverlos.

Para comenzar, es preciso reconsiderar lo que es exactamente un algoritmo «eficaz». En la mayoría de las aplicaciones que se han presentado, ha sido habitual pensar que para que a un algoritmo se le considere eficaz debe ser lineal o ejecutarse en un tiempo proporcional a algo como $N \log N$ o $N^{3/2}$. En general, aquí se ha considerado que los algoritmos cuadráticos son malos y los cúbicos peores. Pero cualquier científico que trabaje con computadoras podría sentirse absolutamente maravillado al conocer algún algoritmo cúbico para cualquiera de los problemas que se van a considerar en este capítulo y en el siguiente. En efecto, incluso un algoritmo en N^{50} pudiera ser agradable (desde un punto de vista teórico), porque estos problemas necesitan un tiempo *exponencial*.

Supóngase que se tiene un algoritmo que lleva un tiempo proporcional a 2^N . Si se tuviera una computadora mil veces más rápida que la más rápida de las supercomputadoras actuales, quizá pudiera resolverse un problema para $N = 50$ en una hora (con la hipótesis más favorable sobre la simplicidad del algoritmo). Pero en dos horas podría hacerse sólo para $N = 51$, y en un año sólo para $N = 59$. E incluso si se desarrollara una nueva computadora con una velocidad un millón de veces mayor, y se tuviera un millón de estas computadoras, no se podría resolver para $N = 100$ en un año. De forma realista, hay que establecer para N un orden de 25 a 30. Un algoritmo «más eficaz» en esta situación debe ser uno que permita resolver un problema para $N = 100$ con una cantidad de tiempo y dinero razonables.

El problema de este tipo más conocido es el *problema del vendedor ambulante*: dado un conjunto de N ciudades, hallar el camino más corto que conecta a todas ellas, sin que se visite ninguna ciudad dos veces. Este problema aparece de forma natural en un número importante de aplicaciones, por lo que se ha

estudiado detenidamente. En este capítulo se utilizará como un ejemplo para examinar algunas técnicas fundamentales. Se han desarrollado muchos métodos avanzados para este problema, pero todavía es impensable poder resolver un ejemplo cualquiera del problema para $N = 1.000$.

El problema del vendedor ambulante es difícil, ya que parece que no hay forma de evitar el tener que comprobar la longitud del gran número de rutas posibles. El verificar cada una de las rutas es una *búsqueda exhaustiva*: primero se verá cómo se hace esto y luego cómo modificar este procedimiento para reducir el número de posibilidades a comprobar, tratando de descubrir decisiones incorrectas lo antes posible durante el proceso de toma de decisiones.

Como se mencionó anteriormente, es imposible resolver un gran problema del vendedor ambulante, desde un punto de vista práctico, aun con las mejores técnicas conocidas. Se verá en el próximo capítulo que lo mismo es cierto para muchos otros importantes problemas técnicos. Pero ¿qué se puede hacer cuando estos problemas aparecen en la práctica? Se desea tener algún tipo de respuesta (el vendedor ambulante tiene que hacer su trabajo): no se puede simplemente ignorar la existencia del problema o decir que es demasiado difícil para ser resuelto. Al final de este capítulo, se verán ejemplos de algunos métodos desarrollados para hacer frente a los problemas prácticos que parecen necesitar una búsqueda exhaustiva. En el capítulo siguiente se examinarán con algún detalle las razones por las que no se pueden encontrar algoritmos eficaces para muchos problemas de este tipo.

Búsqueda exhaustiva en grafos

Si el vendedor ambulante sólo puede viajar entre ciertos pares de ciudades (por ejemplo, si viaja por avión), entonces el problema se modela directamente con un grafo: dado un grafo ponderado (posiblemente dirigido), se desea hallar el ciclo simple más corto que conecta a todos los nodos.

Esto lleva inmediatamente a pensar en otro problema que pudiera resultar más sencillo: dado un grafo no dirigido, ¿existe *alguna* manera de conectar todos los nodos con un ciclo simple? Esto es, comenzando en algún nodo, ¿se puede «visitar» el resto de los nodos y volver al nodo original, visitando cada nodo del grafo solamente una vez? Esto se conoce como el problema del *ciclo de Hamilton*. En el capítulo siguiente se verá que esto es equivalente, en términos de cálculo, al problema del vendedor ambulante, en un sentido estrictamente técnico.

En los capítulos 29 y 31 se vio un cierto número de métodos para visitar sistemáticamente todos los nodos de un grafo. En ellos fue posible organizar los cálculos de forma que cada nodo se visite sólo una vez y se hizo con algoritmos muy eficaces. Para el problema del ciclo de Hamilton ese tipo de solución no está clara: al parecer es necesario visitar cada nodo muchas veces. Para los otros problemas, se construye un árbol, y cuando se alcanza en la búsqueda el «final

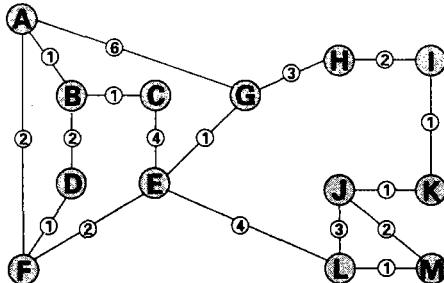


Figura 44.1 Un ejemplo del problema del vendedor ambulante.

de una rama», se puede comenzar desde arriba de nuevo, buscando en otra parte del árbol. Para este problema, el árbol debe tener una estructura particular (un ciclo): si se descubre durante la búsqueda que el árbol que se está construyendo no puede ser un ciclo, es preciso retroceder y reconstruir parte del mismo.

Para ilustrar algunas de las dificultades que se presentan, se examinará el problema del ciclo de Hamilton sobre el grafo de la Figura 44.1. Una búsqueda en profundidad explora los nodos de este grafo en el orden A B C E F D G H I K H L M (suponiendo una representación por matriz de adyacencia o lista de adyacencia). Esto no es un ciclo simple, y por ello para encontrar un ciclo de Hamilton hay que intentar otra forma de visitar los nodos. Es posible tratar todas las posibilidades sistemáticamente con una sencilla modificación del procedimiento *visitar*, como sigue:

```
visitar(int k)
{
    int t;
    val [k] = ++id;
    for (t = 1; t <= V; t++)
        if (a[k][t])
            if (val[t] == 0) visitar(t);
    id--; val[k] = 0;
}
```

En lugar de marcar cada nodo examinado con una entrada *val* diferente de cero, este procedimiento «limpiá despues de pasar» y deja *id* y el array *val* exactamente como los encontró. Los únicos nodos marcados son aquellos para los que *visitar* no ha terminado y éstos corresponden exactamente a un camino simple de longitud *id*, desde el nodo inicial hasta el que se está visitando actualmente. Para *visitar* un nodo, simplemente se visitan todos los adyacentes no

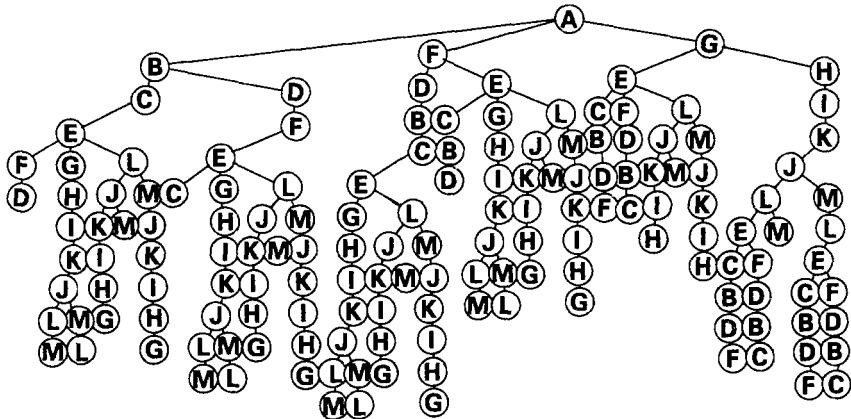


Figura 44.2 Búsqueda exhaustiva.

marcados (los marcados no corresponderían a un camino simple). El procedimiento recursivo comprueba todos los caminos simples del grafo que comienzan en el nodo inicial.

La Figura 44.2 muestra el orden en el que se comprueban los caminos por el procedimiento anterior para el grafo del ejemplo de la Figura 44.1. Cada nodo del árbol corresponde a una llamada a visitar, así que los descendientes de cada nodo son los nodos adyacentes que están no marcados en el momento de la llamada.

Cada camino del árbol entre un nodo y la raíz corresponde a un camino simple del grafo. El primer camino comprobado es A B C E F D; en este punto, todos los vértices adyacentes a D están marcados (tienen una entrada val no nula), y visitar D termina desmarcando a D y volviendo a la llamada anterior. De igual forma, visitar F termina desmarcando a F y volviendo. Luego visitar E intenta con G, que a su vez intenta con H y así sucesivamente hasta que se obtiene el camino A B C E G H I K J L M. Se observa que en la búsqueda en profundidad los nodos permanecen marcados después de haber sido visitados, pero en la búsqueda exhaustiva los nodos se visitan muchas veces. El «desmarcado» de los nodos hace a la búsqueda exhaustiva diferente de la búsqueda en profundidad (aun cuando el código es bastante similar); el lector debería estar seguro de entender esta distinción.

Como se mencionó anteriormente, id es la longitud actual del camino que se recorre y val[k] es la posición del nodo k en ese camino. Por ello se puede hacer que el procedimiento compruebe la existencia de un ciclo de Hamilton haciendo la prueba de que existe una arista desde k a 1 cuando val[k]=V. En el ejemplo anterior, existe sólo un ciclo de Hamilton, que aparece dos veces en el árbol, recorriéndolo en ambas direcciones (existen otros tres caminos de longitud V). El programa puede servir para resolver el problema del vendedor am-

bulante memorizando la longitud del camino actual en el array `val`, y además la longitud mínima de todos los ciclos de Hamilton descubiertos.

Vuelta atrás

El tiempo necesario para ejecutar un procedimiento de búsqueda exhaustiva como el anterior es proporcional al número de llamadas a `visitar`, es decir, al número de nodos del árbol de búsqueda exhaustiva. Para grafos grandes, esto llevará evidentemente mucho tiempo. Por ejemplo, si el grafo es completo (cada nodo está conectado a todos los otros), existen $V!$ ciclos simples, uno por cada organización de los nodos. (Este caso se estudia con más detalle posteriormente.) Incluso para el grafo de la Figura 44.1, no es fácil para un ser humano encontrar un ciclo de Hamilton por simple inspección, de forma que hay que encontrar un método más eficaz de hacerlo por medio de una computadora.

A continuación se examinarán algunas técnicas para reducir sensiblemente el número de posibilidades tratadas. Todas ellas implican añadir pruebas a `visitar` para descubrir las llamadas recursivas que son inútiles para ciertos nodos. Esto corresponde a *podar* el árbol de búsqueda exhaustiva, cortando ciertas ramas y suprimiendo todo lo conectado a ellas.

Una técnica de poda importante es la supresión de simetrías, cuyo valor se manifiesta en el ejemplo anterior por el hecho de que cada ciclo se recorre en ambas direcciones, y por ello se descubrirá dos veces. En este caso se puede asegurar que se descubre cada ciclo sólo una vez obligando a que los conjuntos de tres nodos aparezcan en un orden determinado. Por ejemplo, si se obliga a que el nodo C aparezca después del A, pero antes del B, entonces no hay que llamar a `visitar` para el nodo B a menos que el nodo C esté ya en el camino. Esto conduce al árbol, drásticamente más pequeño, que se muestra en la Figura 44.3.

Esta técnica no es siempre aplicable. Si se supone, por ejemplo, que se trata de encontrar un camino (no necesariamente un ciclo) que conecta a todos los vértices, no se puede utilizar la estrategia anterior, ya que es imposible saber por adelantado si un camino llevará a un ciclo o no.

Cada vez que se interrumpe la búsqueda de un nodo, se evita la búsqueda en el subárbol del que es raíz. Para árboles muy grandes, esto es un ahorro de tiempo sustancial, y realmente la ganancia es tan importante que merece la pena hacer todo lo posible en `visitar` para evitar hacer llamadas recursivas. Existen varias formas de proceder en el caso del ejemplo: una es observar que algunos caminos pueden dividir al grafo de tal forma que los nodos no marcados no están conectados, por lo que pudiera no encontrarse ciclos. Por ejemplo, no existe ningún camino simple que comience por ABE en la Figura 44.1, ya que este camino separa B, C y D del resto del grafo. Por el coste de una búsqueda en profundidad para descubrir esto, pueden evitarse 25 llamadas recursivas a `visitar` (ver la Figura 44.3).

La Figura 44.4 muestra el árbol de búsqueda que resulta cuando se aplica

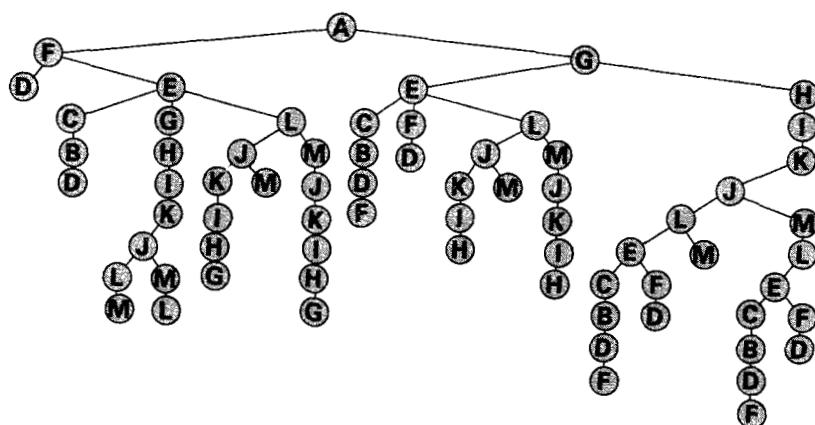


Figura 44.3 Búsqueda de un ciclo con A antes de C y C antes de B.

esta regla al árbol de la Figura 44.3. De nuevo el árbol es drásticamente menor: tiene sólo 19 nodos en comparación con los 153 del árbol completo de búsqueda exhaustiva (Figura 44.2). Es importante notar que las ganancias alcanzadas por este problema en miniatura son sólo indicativas de la situación para problemas mayores. Una poda alta puede llevar a ahorros significativos; el olvido de podas evidentes puede llevar a gastos verdaderamente importantes.

El procedimiento general antes descrito para resolver un problema generando sistemáticamente todas las soluciones posibles se denomina *vuelta atrás*.

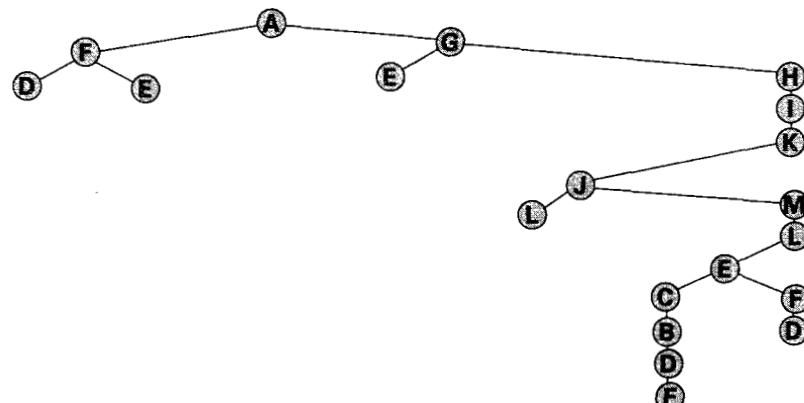


Figura 44.4 Búsqueda de un ciclo con poda cuando se divide el grafo.

Siempre que las soluciones parciales a un problema pueden ampliarse de muchas formas para producir una solución completa, una implementación recursiva como el programa anterior debe ser apropiada. Como anteriormente, el proceso puede ser descrito por un árbol de búsqueda exhaustiva cuyos nodos corresponden a las soluciones parciales. El descenso por el árbol corresponde a un progreso hacia una solución más completa; el ascenso corresponde a una vuelta atrás hacia alguna solución parcial generada previamente, desde la que posiblemente merezca la pena marchar de nuevo hacia adelante.

Como otro ejemplo, se considerará el problema de la mochila del Capítulo 42. Como se mencionó entonces, este problema es considerablemente más difícil cuando los valores no son necesariamente enteros. Para este problema, las soluciones parciales son evidentemente alguna selección de objetos para la mochila, y el proceso de vuelta atrás corresponde a quitar objetos para comprobar alguna otra combinación. La poda del árbol de búsqueda eliminando las simetrías es bastante eficaz para este problema, ya que el orden en el que se colocan los objetos dentro de la mochila no afecta al coste.

Cuando se busca el *mejor* camino (el problema del vendedor ambulante), se dispone de otra técnica de poda importante que termina la búsqueda tan pronto se determina que es imposible tener éxito. Supóngase que se ha encontrado un camino de coste x a través del grafo. Entonces es inútil continuar a lo largo de cualquier otro camino cuyo coste es mayor que x . Esto se puede implementar simplemente no haciendo llamadas recursivas a visitar si el coste del camino parcial actual es mayor que el del mejor camino completo encontrado hasta ahora. Aplicando esta estrategia es claramente imposible eliminar el camino de coste mínimo.

La poda es más efectiva si durante la búsqueda se descubre pronto el camino de menor coste; una forma de provocar esto es visitar los nodos adyacentes al actual por orden creciente del coste. De hecho se puede hacerlo aún mejor: a menudo, es posible calcular una cota del coste de todos los caminos completos que comienzan por un camino parcial dado. Para el ejemplo, se puede obtener una mejor cota del coste de cualquier camino completo que comienza por el camino parcial compuesto por los nodos marcados, añadiendo los costes del árbol de expansión mínimo de los nodos no marcados. (El resto del camino es un árbol de expansión para los nodos no marcados; su coste no será ciertamente menor que el del árbol de expansión de estos nodos.)

Esta técnica general de calcular cotas relativas a soluciones parciales para limitar el número de soluciones completas a examinar se denomina a veces *branch and bound*. Por supuesto, se aplica siempre que los costes están asociados a los caminos (y que se intenta minimizarlos). Normalmente, el objetivo de tales problemas es eliminar un número importante de posibilidades en la búsqueda de una solución —no es inusual aplicar docenas de reglas heurísticas como las descritas en los párrafos anteriores para evitar recorrer el camino equivocado.

La vuelta atrás y branch and bound son muy utilizados como técnicas generales de resolución de problemas. Por ejemplo, constituyen la base de muchos programas de juegos tales como el ajedrez o las damas. En este caso, una

solución parcial es alguna posición legal de todas las piezas en el tablero, y el descendiente de un nodo en el árbol de búsqueda exhaustiva es una posición que puede ser el resultado de algún movimiento legal. El ideal sería que un programa pudiera buscar exhaustivamente a través de todas las posibilidades y seleccionar un movimiento que lleve a la victoria independientemente de lo que haga el oponente, pero por lo regular hay demasiadas posibilidades para hacerlo, de modo que normalmente se hace una búsqueda con vuelta atrás con alguna regla de poda bastante sofisticada que permita que sólo se examinen las posiciones «interesantes». Las técnicas de búsqueda exhaustiva se utilizan también en otras aplicaciones de inteligencia artificial.

En el capítulo siguiente se verán algunos problemas similares a los que se han estado estudiando, que pueden ser abordados utilizando estas técnicas. Resolver un problema particular requiere el desarrollo de criterios sofisticados que pueden ser utilizados para limitar la búsqueda. Aquí se han dado algunos ejemplos de las diferentes técnicas que se han encontrado para el problema del vendedor ambulante e igualmente se han desarrollado métodos sofisticados para otros problemas importantes.

Sin embargo, por muy sofisticados que sean los criterios, en general es cierto que el tiempo de ejecución de los algoritmos con vuelta atrás permanece exponencial. Sin entrar en detalles, si cada nodo del árbol de búsqueda tiene α hijos, por término medio, y la longitud del camino solución es N , entonces se puede esperar que el número de nodos del árbol sea proporcional a α^N . Cada regla de vuelta atrás corresponde a una reducción del valor de α , el número de opciones en cada nodo. Merece la pena realizar este esfuerzo, ya que una reducción en α aumenta el tamaño del problema que puede resolverse. Por ejemplo, un algoritmo que se ejecuta en un tiempo proporcional a $1,1^N$ puede resolver un problema quizás ocho veces mayor que otro que se ejecuta en un tiempo proporcional a 2^N . Por otra parte, como se ha mencionado, ninguno de los dos lo puede hacer bien para problemas muy grandes.

Digresión: Generación de permutaciones

La escritura de un programa que genere todas las posibles formas de reordenar N elementos diferentes es un problema de cálculo interesante. Un programa simple para este problema de *generación de permutaciones* se puede deducir directamente del programa de búsqueda exhaustiva en grafos que se dio anteriormente. Como se destacó entonces, cuando se ejecuta el programa sobre un grafo completo, debe tratar de visitar los vértices del grafo en todos los órdenes posibles. Se obtienen todas las permutaciones manteniendo a los vértices etiquetados en el orden en el que aparecen en el camino de búsqueda e imprimiendo todas las etiquetas de los vértices cada vez que se encuentra un camino de longitud V , como en el siguiente programa:

```

visitar(int k)
{
    int t;
    val[k] = ++id;
    if (id == V) escribirperm();
    for (t = 1; t <= V; t++)
        if (val[t] == 0) visitar(t);
    id--; val[k] = 0;
}

```

Este programa se deduce del procedimiento anterior eliminando toda referencia a la matriz de adyacencia (ya que todas las aristas están presentes en un grafo completo). El procedimiento `escribirperm` imprime simplemente las entradas del array `val`, lo que hace cada vez que `id=V`, es decir cuando se descubre un camino completo en el grafo. (De hecho se puede mejorar el programa un poco si se omite el bucle `for` cuando `id=V`, ya que en este caso se sabe que todas las entradas de `val` son diferentes de cero.) Para imprimir todas las permutaciones de los enteros desde 1 hasta N , se invoca este procedimiento por medio de la llamada a `visitar(0)` con `id` inicializado a -1 y el array `val` inicializado a cero. Esto corresponde a introducir un nodo ficticio en el grafo completo, y comprobar todos los caminos del grafo comenzando por el nodo 0. Cuando se invoca de esta forma para $N = 4$, el procedimiento produce la salida siguiente (impresa aquí en dos columnas):

1 2 3 4	2 3 1 4
1 2 4 3	2 4 1 3
1 3 2 4	3 2 1 4
1 4 2 3	4 2 1 3
1 3 4 2	3 4 1 2
1 4 3 2	4 3 1 2
2 1 3 4	2 3 4 1
2 1 4 3	2 4 3 1
3 1 2 4	3 2 4 1
4 1 2 3	4 2 3 1
3 1 4 2	3 4 2 1
4 1 3 2	4 3 2 1

Por supuesto, la interpretación del procedimiento de generación de caminos en un grafo completo es patentemente visible. Pero un análisis directo del procedimiento muestra que genera todas las $N!$ permutaciones de los enteros entre 1 y N generando primero las $(N - 1)!$ permutaciones con 1 en primera posición (llamándose a sí mismo recursivamente desde el 2 hasta el N), y generando después las $(N - 1)!$ permutaciones con 1 en segunda posición, etcétera.

Sería impensable utilizar este programa incluso para $N = 16$, ya que $16! >$

2^{50} . No obstante, es un programa importante para estudiar ya que puede constituir la base de un programa de vuelta atrás para resolver cualquier problema que implique la reorganización de un conjunto de elementos.

Por ejemplo, considérese el *problema euclíadiano del vendedor ambulante*: dado un conjunto de N puntos del plano, hallar el recorrido más corto que los conecta a todos. Como cada ordenamiento de los puntos corresponde a un recorrido legal, el programa anterior puede servir para la búsqueda exhaustiva de la solución a este problema cambiándolo simplemente para memorizar el coste de cada recorrido y el coste mínimo obtenido, como anteriormente. Entonces se puede utilizar la misma técnica de *branch and bound* anterior, así como las diversas heurísticas de vuelta atrás específicas del problema euclíadiano. (Por ejemplo, es fácil demostrar que el recorrido óptimo no puede cruzarse a sí mismo, y de esta forma se puede interrumpir la búsqueda en todos los caminos parciales que se «autocruzan».) A búsquedas heurísticas diferentes corresponden diferentes formas de ordenar las permutaciones. Tales técnicas pueden ahorrar una enorme cantidad de trabajo, pero siempre dejan otra gran cantidad por hacer. No es nada simple hallar una solución exacta al problema euclíadiano del vendedor ambulante, incluso para un N tan pequeño como 16.

Otra razón por la que la generación de permutaciones es de interés, es que existen un número de procedimientos relacionados para la generación de objetos combinatorios. En algunos casos, el número de objetos generados no es tan numeroso como las permutaciones, y tales procedimientos pueden ser útiles en la práctica para valores grandes de N . Un procedimiento para generar todas las formas de elegir un subconjunto de tamaño k en un conjunto de N objetos es un ejemplo de este tipo. Para N grande y k pequeño, el número de formas de hacer esto es aproximadamente proporcional a N^k . Tal procedimiento se podría utilizar como base para un programa de vuelta atrás para resolver el problema de la mochila.

Algoritmos de aproximación

Puesto que encontrar el recorrido más corto parece requerir una gran cantidad de cálculo, es razonable considerar que debe ser más fácil encontrar un recorrido casi tan corto como el más corto. Si se tiene la esperanza de poder aflojar la restricción de que tiene que ser obligatoriamente el camino más corto posible, entonces se puede tratar con problemas más grandes que los permitidos por las técnicas anteriores.

Por ejemplo, es relativamente fácil encontrar un recorrido de longitud inferior al doble de la del recorrido óptimo. El método está basado en encontrar simplemente el árbol de expansión mínimo: esto no sólo proporciona una cota inferior de la longitud del recorrido, como se mencionó anteriormente, sino que también proporciona una *cota superior*, de la siguiente forma. Dado un árbol de expansión mínimo, se puede deducir un recorrido visitando los nodos del

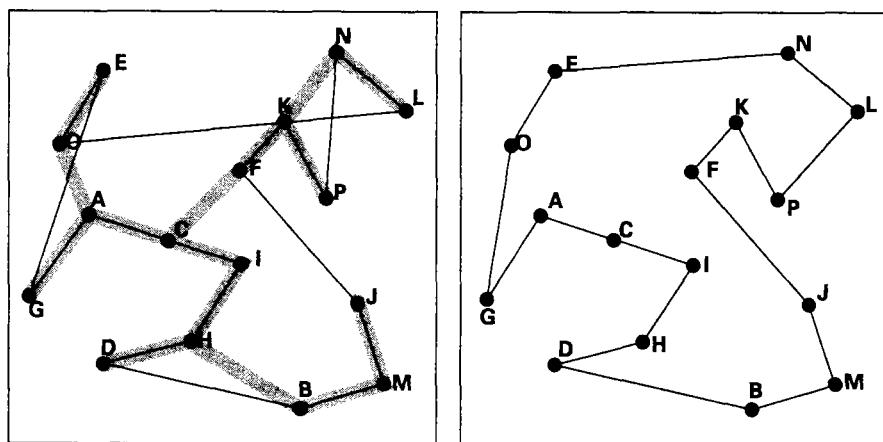


Figura 44.5 Recorrido euclíadiano simple del vendedor ambulante.

árbol de expansión mínimo utilizando el procedimiento siguiente: procesar el nodo x , visitar x , visitar después cada hijo de x , aplicando este procedimiento recursivamente y volviendo al nodo x después de visitar cada uno de sus hijos, terminando en el nodo x . Este recorrido atraviesa dos veces cada arista del árbol de expansión, de modo que el coste es dos veces el del árbol. No es un recorrido simple, ya que un nodo puede ser visitado muchas veces, pero puede convertirse en un recorrido simple eliminando todas las ocurrencias de cada nodo, excepto la primera. Eliminar una ocurrencia de un nodo corresponde a tomar un camino más corto evitando dicho nodo: ciertamente esto no puede incrementar el coste del recorrido, de modo que se obtiene un recorrido simple que tiene un coste inferior al doble del árbol de expansión mínimo.

Por ejemplo, el diagrama de la izquierda de la Figura 44.5 muestra un árbol de expansión mínimo para el conjunto de puntos de muestra (calculado como se describió en el Capítulo 31), y el recorrido simple correspondiente. Por supuesto, este último no es el óptimo, ya que se autocruza. Para un conjunto más grande de puntos aleatorios, parece que el recorrido se aproxima al óptimo, aunque no se ha hecho ningún análisis para apoyar esta conclusión.

Otro enfoque conocido consiste en desarrollar técnicas para mejorar un recorrido ya existente, con la esperanza de que puedan encontrarse recorridos cada vez más cortos aplicando repetidamente dichas técnicas. Por ejemplo, en el problema *euclíadiano* del vendedor ambulante, donde las distancias en el grafo son distancias entre puntos del plano, un recorrido que se autocruza se puede mejorar eliminando cada intersección de la siguiente forma. Si los segmentos AB y CD se cortan, se obtiene un recorrido más corto eliminando AB y CD y añadiendo AD y CB. Aplicando este procedimiento sucesivamente producirá, para cualquier recorrido, uno de menor longitud y que no se cruza a sí mismo. Por ejemplo, el procedimiento aplicado al recorrido obtenido a partir del árbol de

expansión mínimo (a la izquierda de la Figura 44.5) da el recorrido más corto de la derecha de la misma figura.

De hecho, una de las formas más eficaces de obtener soluciones aproximadas al problema euclíadiano del vendedor ambulante, descubierta por S. Lin, es generalizar el procedimiento anterior para mejorar recorridos, intercambiando tres o más aristas de un recorrido que ya existe. Se han obtenido resultados muy buenos aplicando tal procedimiento sucesivamente a un recorrido inicial *aleatorio* hasta que no permita más mejoras. Se podría pensar que sería preferible comenzar con un recorrido que ya estuviera cerca del óptimo, pero los estudios de Lin indican que éste no es siempre el caso.

Los diversos enfoques para producir soluciones aproximadas al problema del vendedor ambulante descritos anteriormente son solamente indicativos de los tipos de técnicas que se pueden utilizar para evitar la búsqueda exhaustiva. Las breves descripciones anteriores no justifican la cantidad de ideas ingeniosas que se han desarrollado: la formulación y el análisis de los algoritmos de este tipo constituyen todavía un área bastante activa de investigación en la informática.

Es lícito preguntarse por qué el problema del vendedor ambulante y los otros problemas que se han estado mencionando *requieren* una búsqueda exhaustiva. ¿No sería posible que un algoritmo ingenioso encontrara el recorrido mínimo tan fácil y rápidamente como se encuentra el árbol de expansión mínimo? En el capítulo siguiente se verá por qué la mayoría de los científicos creen que no existe tal algoritmo y por qué los algoritmos del tipo de los presentados en esta sección deben ser, en consecuencia, objeto de estudio.

Ejercicios

1. ¿Qué es preferible utilizar, un algoritmo que necesita N^5 pasos o uno que necesita 2^N ?
2. ¿El grafo «laberinto» del Capítulo 29 contiene un ciclo de Hamilton?
3. Dibujar el árbol que corresponde a la Figura 44.4 cuando se está buscando un ciclo de Hamilton en el grafo del ejemplo comenzando en el vértice B en lugar del A.
4. ¿Cuánto tiempo llevaría una búsqueda exhaustiva para encontrar un ciclo de Hamilton en un grafo en el que todos los nodos están conectados exactamente a otros dos nodos? Responder a la misma pregunta para el caso en el que todos los nodos están conectados a otros tres nodos.
5. ¿Cuántas llamadas a visitar se hacen (en función de V) por el procedimiento de generación de permutaciones?
6. Encontrar un procedimiento de generación de permutaciones no recursivo a partir del programa dado.
7. Escribir un programa que determine cuándo dos matrices de adyacencia dadas representan o no al mismo grafo, excepto con diferentes nombres de vértices.

8. Escribir un programa para resolver el problema de la mochila del Capítulo 42 cuando las longitudes pueden ser números reales.
9. Escribir un programa para contar el número de árboles de expansión de un conjunto de N puntos del plano con aristas que no se cortan.
10. Resolver el problema euclíadiano del vendedor ambulante para el ejemplo de un conjunto de 16 puntos.

Problemas NP-completos

Por lo regular, los algoritmos que se han estudiado en este libro se utilizan para resolver problemas prácticos y, por consiguiente, consumen una razonable cantidad de recursos. La utilidad práctica de la mayoría de ellos es evidente: para un gran número de problemas puede elegirse entre un gran número de algoritmos eficaces. Por desgracia, como se indicó en el capítulo anterior, en la práctica surgen muchos problemas que no admiten estas soluciones eficaces. Lo que es peor, para una gran clase de problemas no se puede ni siquiera decir cuándo existe o no una solución eficaz.

Este estado de cosas ha sido una causa de gran frustración entre los programadores y diseñadores de algoritmos, que no han podido encontrar un buen algoritmo para una gran variedad de problemas prácticos, así como entre los teóricos, que se han visto imposibilitados de encontrar una razón que explique por qué estos problemas se deben considerar como difíciles. Se han hecho muchas investigaciones en esta área, que han conducido al desarrollo de mecanismos mediante los cuales los nuevos problemas pueden clasificarse como «tan difíciles como» otros problemas viejos desde un punto de vista técnico concreto. Aunque gran parte de este trabajo está fuera del alcance de este libro, las ideas centrales no son difíciles de comprender. Es verdaderamente útil cuando uno se enfrenta a un nuevo problema tener alguna idea de los tipos de problemas para los que nadie conoce un algoritmo eficaz.

Algunas veces la línea que separa los problemas «fáciles» de los «difíciles» es bastante fina. Por ejemplo, en el Capítulo 31 se vio un algoritmo eficaz para el problema siguiente: «Hallar el camino más corto entre el vértice x y el vértice y en un grafo ponderado dado». Pero si se pide el camino *más largo* (sin ciclos) entre x e y , se tiene un problema para el que nadie conoce una solución sustancialmente mejor que la de verificar todos los caminos posibles. Lo tenue de esa línea es aún más evidente cuando se consideran problemas similares que sólo solicitan respuestas «si-no»:

Fácil:

¿Existe un camino entre x e y con peso $\leq M$?

Difícil: ¿Existe un camino entre x e y con peso $\geq M$?

La búsqueda en profundidad conduce a una solución del primer problema en tiempo lineal, pero todos los algoritmos conocidos pueden llevar un tiempo exponencial para el segundo problema. Se podría ser más preciso al decir que «pueden llevar un tiempo exponencial», pero esto no se hará aquí. Generalmente, es útil pensar que un algoritmo de tiempo exponencial es aquel para el que, para alguna entrada de tamaño N , lleva un tiempo proporcional a 2^N (como mínimo). (La esencia de los resultados que se están presentando no cambia si se reemplaza 2 por cualquier número $\alpha > 1$.) Esto significa, por ejemplo, que no se puede garantizar que un algoritmo de tiempo exponencial funcione con problemas de tamaño (por ejemplo) 100 o mayores, ya que nadie puede esperar la ejecución de un algoritmo que lleva 2^{100} pasos, sin tener en cuenta la velocidad de la computadora. El crecimiento exponencial ridiculiza a los cambios tecnológicos: una supercomputadora puede ser un billón de veces más rápida que un ábaco, pero ninguno de los dos puede resolver problemas que necesitan 2^{100} pasos.

Algoritmos deterministas y no deterministas de tiempo polinómico

La gran disparidad de rendimiento entre los algoritmos «eficaces» del tipo de los que se han estado estudiando y los de fuerza bruta «exponencial» que verifican todas las posibilidades, justifica el estudio de la interfaz que los separa con un simple modelo formal. En él, la eficacia de un algoritmo es una función del número de bits utilizados para codificar la entrada, por medio de un esquema de codificación «razonable». (La definición precisa de «razonable» incluye todos los métodos comunes de codificación de información: un ejemplo de esquema de código no razonable es el sistema unario, donde se utilizan M bits para representar el número M . Más bien, podría esperarse que el número de bits utilizados para representar el número M fuera proporcional a $\log M$.) Aquí se desea simplemente identificar todo algoritmo que garantice su ejecución en un tiempo proporcional a algún polinomio del número de bits de la entrada. Todo problema que puede ser resuelto por un tal algoritmo se dice que pertenece a

P: el conjunto de todos los problemas que pueden ser resueltos por algoritmos deterministas en tiempo polinómico.

Por *deterministas* se quiere expresar que, en cualquier momento, no importa lo que esté haciendo el algoritmo, existe sólo una acción que puede hacer después. Esta noción muy general cubre la forma como se ejecutan los programas en las computadoras reales. Obsérvese que el polinomio no se especifica del

todo y que esta definición engloba sin duda a los algoritmos estándar que se han estado estudiando hasta ahora. La ordenación pertenece a P ya que (por ejemplo) la ordenación por inserción se ejecuta en un tiempo proporcional a N^2 (la existencia de algoritmos de ordenación en $N \log N$ no es relevante en el contexto presente). Además, el tiempo que lleva un algoritmo depende evidentemente de la computadora utilizada, pero el empleo de computadoras diferentes afecta al tiempo de ejecución sólo en un factor polinómico (de nuevo, dentro de unos límites razonables), por lo que estas consideraciones no tienen importancia en este contexto.

Por supuesto, los resultados teóricos que se están presentando se basan en un modelo de cálculo completamente específico dentro del cual se pueden demostrar. La intención es examinar algunas de las ideas fundamentales, no desarrollar definiciones rigurosas ni teoremas.

El lector puede estar seguro de que cualquier error lógico se debe a la naturaleza informal de la descripción y no a la teoría por sí misma.

Una forma «no razonable» de extender la potencia de una computadora es conferirle la potencia del *no determinismo*: cuando un algoritmo se enfrenta ante una búsqueda de varias opciones, tiene la capacidad de «adivinar» la correcta. Para las necesidades de la presentación posterior, se puede pensar que un algoritmo para una máquina no determinista es capaz de «adivinar» la solución del problema, y después verificar que es la correcta. En el Capítulo 20 se vio que el no determinismo puede ser útil como una herramienta para diseñar algoritmos; aquí servirá como un dispositivo teórico para ayudar a clasificar los problemas. Sea

NP: el conjunto de todos los problemas que se pueden resolver por algoritmos no deterministas en tiempo polinómico.

Por supuesto, todo problema de P está también en NP. Pero parece como si existieran muchos otros problemas en NP: para demostrar que un problema está en NP, es suficiente con encontrar un algoritmo de tiempo polinómico para comprobar que una solución dada (la solución adivinada) es válida. Por ejemplo, la versión «sí-no» del problema del camino más largo está en NP, así como el problema de la *satisfactibilidad*. Dada una fórmula lógica de la forma

$$(x_1 + x_3 + x_5) * (x_1 + \bar{x}_2 + x_4) * (\bar{x}_3 + x_4 + x_5) * (x_2 + \bar{x}_3 + x_5)$$

donde los x_i representan variables booleanas (**verdadero** o **falso**), «+» representa **or**, «*» representa **and** y \bar{x} representa **not**, el problema de la satisfactibilidad consiste en determinar cuándo existe o no una asignación de valores verdaderos a las variables que hacen la fórmula **verdadera** (la «**satisface**»). Se verá más adelante que este problema particular tiene un papel especial en la teoría.

El no determinismo es una operación de tal potencia que parece casi absurdo considerarla seriamente. ¿Por qué preocuparse por una herramienta imaginaria que hace que los problemas difíciles parezcan triviales? La respuesta es

que, por muy poderoso que pueda parecer el no determinismo, ¡nadie ha sido capaz de probar que sea una ayuda para algún problema particular! Dicho de otra forma, nadie ha sido capaz de encontrar un solo problema que se pueda demostrar que está en NP, pero no está en P (o incluso probar que existe uno): no se sabe si $P = NP$ (o no). Ésta es una situación bastante frustrante, ya que muchos problemas prácticos importantes pertenecen a NP (se podrían resolver eficazmente por una máquina no determinista), pero pueden o no pertenecer a P (no se conoce ningún algoritmo eficaz para ellos en una máquina determinista). Si se pudiera probar que un problema no pertenece a P, entonces se podría abandonar la búsqueda de un algoritmo eficaz que lo resuelva. En ausencia de tal prueba, existe la posibilidad de que no se haya descubierto algún algoritmo eficaz. En efecto, dado el estado actual del conocimiento, podría existir algún algoritmo eficaz para *todos* los problemas de NP, lo que querría decir que no se han descubierto muchos algoritmos eficaces. En realidad, nadie cree que $P = NP$, y se han practicado esfuerzos considerables para demostrar lo contrario, pero este punto permanece como uno de los problemas de investigación abiertos y excepcionales de la informática.

Compleción NP

Más adelante se verá un conjunto de problemas que se sabe que pertenecen a NP, pero que pueden o no pertenecer a P. Esto es, son fáciles de resolver en una máquina no determinista, pero, a pesar de un considerable esfuerzo, nadie ha sido capaz de encontrar un algoritmo eficaz sobre una máquina convencional (o probar que no existe ninguno) para alguno de ellos. Estos problemas tienen una propiedad suplementaria que es una prueba convincente de que $P \neq NP$: si *alguno* de los problemas puede ser resuelto en un tiempo polinómico en una máquina determinista, entonces pueden *serlo todos* los otros (es decir $P = NP$). Esto es, el fracaso colectivo de todos los investigadores al no encontrar algoritmos eficaces para todos estos problemas puede ser visto como el proceso colectivo al no probar que $P = NP$. Tales problemas se dicen que son *NP-completos*. Se ha encontrado que un gran número de problemas prácticos interesantes tienen esta característica.

La técnica esencial utilizada para demostrar que un problema es NP-completo consiste en la idea de la *reducción polinómica*. Se demuestra que cualquier algoritmo que resuelva un nuevo problema en NP puede utilizarse para resolver cualquier problema conocido NP-completo por el siguiente proceso: se transforma un elemento arbitrario del problema conocido NP-completo en un elemento del nuevo problema, se resuelve el problema utilizando el algoritmo dado, y después se transforma la solución a la inversa en una solución del problema NP-completo. Se vio un ejemplo de un proceso similar en el Capítulo 34, al reducir la concordancia de un grafo bipartido en el flujo de red. Por «reducible

polinómicamente» se entiende que las transformaciones se pueden hacer en tiempo polinómico: así, la existencia de un algoritmo de tiempo polinómico para el nuevo problema podría implicar la existencia de un algoritmo de tiempo polinómico para el problema NP-completo, y esto podría (por definición) implicar la existencia de algoritmos de tiempo polinómico para todos los problemas en NP.

El concepto de reducción proporciona un mecanismo útil para la clasificación de algoritmos. Por ejemplo, para probar que un problema en NP es NP-completo, es suficiente con mostrar que algún problema NP-completo conocido es reducible polinómicamente a aquél: esto es, que un algoritmo de tiempo polinómico para el nuevo problema se puede utilizar para resolver el problema NP-completo, y entonces puede, de hecho, utilizarse para resolver todos los problemas NP. Como ejemplo de reducción, se recuerdan los dos problemas del Capítulo 44:

VENDEDOR AMBULANTE: Dado un conjunto de ciudades y de distancias entre todos los pares de ellas, hallar un recorrido de todas las ciudades de distancia menor que M .

CICLO DE HAMILTON: Dado un grafo, encontrar un ciclo simple que contenga todos los vértices.

Supóngase que se sabe que el problema del ciclo de Hamilton es NP-completo, y se desea determinar si o no el problema del vendedor ambulante es también NP-completo. Cualquier algoritmo para resolver el problema del vendedor ambulante puede utilizarse para resolver el problema del ciclo de Hamilton, por medio de la siguiente reducción: dado un elemento del problema del ciclo de Hamilton (un grafo), construir otro del problema del vendedor ambulante (un conjunto de ciudades, con las distancias entre todos los pares) de la siguiente forma: para las ciudades del vendedor ambulante se utiliza el conjunto de vértices del grafo; para las distancias entre cada dos ciudades se utiliza 1 si existe una arista entre los vértices correspondientes del grafo y 2 si no existe. Entonces se tiene el algoritmo para el problema del vendedor ambulante: hallar un recorrido de distancia menor o igual a N , el número de vértices del grafo. Este recorrido debe corresponder precisamente al ciclo de Hamilton. Un algoritmo eficaz para el problema del vendedor ambulante sería también un algoritmo eficaz para el problema del ciclo de Hamilton. Esto es, el problema del ciclo de Hamilton se reduce al problema del vendedor ambulante, de modo que la complejidad NP del problema del primero implica la del segundo.

La reducción del problema del ciclo de Hamilton al del vendedor ambulante es relativamente simple, ya que los problemas son similares. En realidad, las reducciones polinómicas pueden ser bastante complicadas y conectar problemas que pudieran parecer bastante diferentes. Por ejemplo, es posible reducir el problema de satisfactibilidad al problema del ciclo de Hamilton. Sin entrar en detalles, se obtendrá un esbozo de la demostración.

Lo que se quiere demostrar es que si se tenía una solución de tiempo poli-

nómico para el problema del ciclo de Hamilton, entonces se podría obtener una del mismo tipo polinómico para el problema de la satisfactibilidad, por medio de una reducción polinómica. La prueba consiste en elaborar un método detallado de construcción que va mostrando cómo, dado un elemento del problema de satisfactibilidad (una fórmula booleana), construir (en tiempo polinómico) otro del problema del ciclo de Hamilton (un grafo) con la propiedad de que conociendo cuándo el grafo tiene un ciclo de Hamilton se sabe cuándo la fórmula es satisfacible. El grafo se construye a partir de componentes pequeñas (correspondientes a las variables) que pueden ser atravesadas por un camino simple dirigido (según el valor verdadero o falso de las variables). Estas pequeñas componentes están enlazadas entre sí como especifican las cláusulas, utilizando subgrafos más complicados que pueden ser atravesados por caminos simples que corresponden a los valores verdadero o falso de las variables. Hay un abismo entre esta breve descripción y la construcción completa, y el desarrollo de pruebas rigurosas y detalladas de este tipo es bastante complicado (aunque los expertos en este campo acaban siendo también expertos en construir tales «inveniones» para las reducciones). La idea aquí es mostrar que la reducción polinómica puede aplicarse a problemas bastante más diferentes.

Así que, si se dispusiera de un algoritmo de tiempo polinómico para el problema del vendedor ambulante, entonces se podría tener también un algoritmo de tiempo polinómico para el del ciclo de Hamilton, que daría, a su vez, un algoritmo del mismo tipo para el de la satisfactibilidad. Cada problema que se demuestra que es un NP-completo proporciona otra base potencial para probar otros futuros problemas NP-completos. La demostración podría ser tan simple como la reducción anterior del problema del ciclo de Hamilton al del vendedor ambulante, o tan complicada como la transformación del problema de la satisfactibilidad al del ciclo de Hamilton, o estar a medio camino. Literalmente, se ha demostrado que miles de problemas de una amplia gama de áreas de aplicación son NP-completos al transformar uno en otro de esta forma.

El teorema de Cook

La reducción utiliza la completitud NP de un problema para deducir otro. Existe un caso, no obstante, en el que la reducción no se puede utilizar: ¿cómo se demostró por primera vez que un problema era NP-completo? Es lo que hizo S. A. Cook en 1971 al dar una prueba directa de que la satisfactibilidad es NP-completa: esto es, si existe un algoritmo de tiempo polinómico para la satisfactibilidad, entonces todos los problemas en NP pueden ser resueltos en tiempo polinómico.

La demostración es extremadamente complicada, pero se puede explicar el método general. Primero, se desarrolla una definición matemática completa de una máquina capaz de resolver cualquier problema en NP. Se trata de un modelo simple de computadora de propósito general conocida como la *máquina*

de Turing, que puede leer entradas, ejecutar ciertas operaciones y escribir salidas. Una máquina de Turing puede ejecutar cualquier cálculo que cualquier otra computadora de propósito general pueda calcular, utilizando la misma cantidad de tiempo (dentro de un factor polinómico), y tiene la ventaja adicional de tener una descripción matemáticamente concisa. Dotada con la potencia adicional del no determinismo, la máquina de Turing puede resolver cualquier problema en NP. El paso siguiente de la demostración es describir cada característica de la máquina, incluyendo cómo se ejecutan las instrucciones, en términos de fórmulas lógicas tales como aparecen en el problema de la satisfactibilidad. De esta forma se establece una correspondencia entre cada problema en NP (que se puede expresar como un programa en la máquina no determinista de Turing) y algún elemento de la satisfactibilidad (la traducción de este programa a una fórmula lógica). Ahora, la solución al problema de la satisfactibilidad corresponde esencialmente a una simulación de la máquina ejecutando un programa dado con una entrada dada, para que produzca una solución del elemento del problema dado. Los detalles suplementarios de esta demostración están fuera del alcance de este libro. Por fortuna, en realidad sólo hace falta una prueba: es mucho más fácil utilizar la reducción para probar la completitud NP.

Algunos problemas NP-completos

Como se mencionó anteriormente, se sabe que miles de problemas diversos son NP-completos. En esta sección, se mencionarán algunos para ilustrar la extensión de problemas que se han estudiado. Por supuesto, la lista comienza con la *satisfactibilidad* e incluye *el vendedor ambulante* y *el ciclo de Hamilton*, así como *el camino más largo*. Los problemas siguientes son representativos:

PARTICIÓN: Dado un conjunto de enteros, ¿es posible dividirlos en dos conjuntos con la misma suma?

PROGRAMACIÓN LINEAL EN ENTEROS: Dado un programa lineal, ¿tiene una solución entera?

PLANIFICACIÓN DEL MULTIPROCESADOR: Dada una fecha de entrega y un conjunto de tareas de longitud diferente para ser ejecutadas en dos procesadores idénticos, ¿pueden ordenarse las tareas de forma tal que se cumpla la fecha de entrega?

RECUBRIMIENTO DE VÉRTICES: Dado un grafo y un entero N , ¿existe un conjunto de como máximo N vértices en el que se toquen todas las aristas?

Estos y otros muchos problemas tienen importantes aplicaciones prácticas, y ha habido una fuerte motivación durante años para encontrar buenos algoritmos que los resuelvan. El hecho de que no se pueda encontrar un buen algoritmo para alguno de estos problemas es seguramente una evidencia de que $P \neq NP$,

y la mayoría de los investigadores creen ciertamente que este es el caso. (Por otro lado, el hecho de que nadie ha sido capaz de probar que cualquiera de estos problemas no pertenece a P podría utilizarse como una evidencia en sentido contrario). Sea o no P = NP, el hecho práctico es que en estos momentos no se cuenta con algoritmos capaces de resolver cualquier problema NP-completo con eficacia.

Como se indicó en el capítulo anterior, se han desarrollado algunas técnicas para combatir esta situación, puesto que es preciso encontrar algún tipo de solución a estos problemas en la práctica. Un enfoque es modificar el problema y encontrar un algoritmo de «aproximación» que encuentre no la mejor solución, pero que garantiza una próxima a la mejor. (Por desgracia, algunas veces esto no es suficiente para evitar la compleción NP.) Otro enfoque es contar con un rendimiento de «tiempo medio» y desarrollar un algoritmo que encuentre la solución en algunos casos, pero que no funciona necesariamente en todos. Esto es, mientras que puede no ser posible encontrar un algoritmo que garantice un buen funcionamiento en todos los elementos de un problema, pudiera ser posible resolver con eficacia todos los problemas que virtualmente surgen en la práctica. Un tercer enfoque es trabajar con algoritmos exponenciales «eficaces», utilizando las técnicas de vuelta atrás descritas en el capítulo anterior. Por último, existe una brecha bastante grande entre el tiempo polinómico y el exponencial que no ha sido explorado por la teoría. ¿Qué se puede decir de un algoritmo que se ejecute en un tiempo proporcional a $N^{\log N}$ o $2^{\sqrt{N}}$?

Todas las áreas de aplicación que se han estudiado en este libro están tocadas por la compleción NP: existen problemas NP-completos en aplicaciones numéricas, en ordenación y búsqueda, en procesamiento de cadenas, en geometría, y en procesamiento de grafos. La contribución práctica más importante de la teoría de la compleción NP es que proporciona un mecanismo para descubrir cuando un nuevo problema de cualquiera de estas áreas es «fácil» o «difícil». Si se puede encontrar un algoritmo eficaz para resolver un problema nuevo, entonces no existe dificultad. Si no, la demostración de que el problema es NP-completo indica al menos que desarrollar un algoritmo eficaz podría ser un avance extraordinario (y sugiere que debería adoptarse un enfoque diferente). La gran cantidad de algoritmos eficaces que se han examinado en este libro es el testimonio de que se ha aprendido bastante sobre métodos de cálculo desde Euclides, pero la teoría de compleción NP muestra que, verdaderamente, aún hay mucho que aprender.

Ejercicios

1. Escribir un programa para encontrar el camino más largo entre x e y en un grafo ponderado dado.
2. ¿Puede existir un algoritmo que resuelva un problema NP-completo en un tiempo *medio* de $M \log N$, si $P \neq NP$? Explicar la respuesta.

3. Desarrollar un algoritmo de tiempo polinómico no determinista para resolver el problema de la partición.
4. ¿Existe una reducción de tiempo polinómico inmediata entre el problema del vendedor ambulante sobre grafos y el mismo problema pero euclíadiano, y viceversa?
5. ¿Cuál sería el significado de un programa que podría resolver el problema del vendedor ambulante en un tiempo proporcional a $1,1^N$?
6. ¿Es la fórmula lógica dada en el texto satisfactible?
7. ¿Podría utilizarse una de las «máquinas algoritmos» con paralelismo completo para resolver un problema NP-completo en tiempo polinómico, si $P \neq NP$? Explicar la respuesta.
8. ¿Dónde se sitúa el problema «calcular el valor exacto de 2^N » en el modelo de clasificación P-NP?
9. Demostrar que el problema de encontrar un ciclo de Hamilton en un grafo *dirigido* es NP-completo, utilizando la compleción NP del problema del ciclo de Hamilton para grafos no dirigidos.
10. Supóngase que se sabe que dos problemas son NP-completos. ¿Esto implica que existe una reducción de tiempo polinómico de uno al otro, si $P \neq NP$?

REFERENCIAS para Temas avanzados

Cada uno de los temas cubiertos en esta sección es el contenido de varios volúmenes de material de referencia. El lector que desee más información debe anticipar que tendrá que ponerse a estudiar en serio; aquí sólo se puede brindar algunas referencias básicas.

La máquina de la mezcla perfecta del Capítulo 40 se describe en el artículo de Stone, que cubre muchas otras aplicaciones. Se encontrará más información sobre los arrays sistólicos de Kung y Leiserson en el libro de Mead y Conway sobre VLSI. El texto estándar sobre procesamiento de señales digitales y la FFT es el libro de Oppenheim y Schafer. Se pueden encontrar más detalles sobre programación dinámica (y temas de otros capítulos) en el libro de Hu. El tratamiento de la programación lineal del Capítulo 43 se basa en la excelente descripción del libro de Papadimitriou y Steiglitz, en el que todos los argumentos intuitivos se respaldan con pruebas matemáticas completas. Una mayor información sobre las técnicas de búsqueda exhaustiva puede encontrarse en los libros de Wells y Reingold, Nievergelt y Deo. Finalmente, el lector interesado en más información sobre la complejidad NP puede consultar el artículo de Lewis y Papadimitriou y el libro de Garey y Johnson, que contiene una descripción amplia de varios tipos de complejidad NP y una lista de cientos de problemas NP-completos.

- M. R. Garey y D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
- T. C. Hu, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.
- H. R. Lewis y C. H. Papadimitriou, «The efficiency of algorithms», *Scientific American*, 238, 1 (1978).
- C. A. Mead y L. C. Conway, *Introduction to VLSI Design*, Addison-Wesley, Reading, MA, 1980.
- C. H. Papadimitriou y K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
- E. M. Reingold, J. Nievergelt y N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
- A. V. Oppenheim y R. W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975.
- H. S. Stone, «Parallel processing with the perfect shuffle», *IEEE Transactions on Computing*, C-20, 2 (febrero, 1971).
- M. B. Wells, *Elements of Combinatorial Computing*, Pergamon Press, Oxford, 1971.

Epílogo

Los algoritmos de este libro han tenido al menos una aplicación: producir el propio libro. En gran medida, cuando el texto dice «el programa anterior genera la figura siguiente», literalmente es verdad. El libro se ha realizado por medio de un sistema de fotocomposición informatizado y la mayoría de los gráficos se generaron automáticamente por los programas que aparecen en él. La razón esencial de organizar el trabajo de esta manera es que permite que se generen fácilmente gráficos complejos; una consecuencia benéfica es que se puede tener cierta confianza en que los programas trabajarán de la forma prometida para otras aplicaciones. Esta aproximación ha sido posible gracias a los recientes avances en la industria de la impresión y por la juiciosa utilización de modernos sistemas y programas de composición.

El libro está constituido por más de tres mil archivos de computadora, al menos uno por cada figura y programa y por cada capítulo de texto. La composición del libro comprende no solamente el trabajo normal de situar los caracteres en el texto, sino también la ejecución de programas bajo el control de los archivos de figuras, para la producción de descripciones de alto nivel de las figuras que posteriormente pueden imprimirse. Este proceso se describe brevemente a continuación.

Cada algoritmo está implementado en C++, C y Pascal. Los programas son archivos independientes, escritos de tal forma que puedan ser ejecutados por programas piloto para su depuración, o puedan ser introducidos en el texto para su impresión. En el texto se puede hacer referencia a un programa directamente por su contenido, en cuyo caso se ejecuta en un filtro de formateo; o se puede hacer referencia indirectamente (por medio de un archivo de figura) para las salidas que produzca, en cuyo caso se ejecuta y sus salidas se dirigen hacia los programas de diseño que dibujan las figuras. Durante la depuración por lo regular se simplificó el programa de salida, como se describe posteriormente, aunque algunos errores fueron más fáciles de observar en las propias figuras.

La interfaz entre los programas y el software de diseño responde a un modelo del método desarrollado por Marc Brown y el autor en el marco de un sistema interactivo que proporciona una visualización dinámica de los algoritmos al ejecutarse en aplicaciones para la educación y otros temas. Los algoritmos se han instrumentado para producir «acontecimientos interesantes» en momentos importantes de la ejecución y así proporcionar informaciones relativas a las modificaciones de las estructuras de datos subyacentes. Asociado con cada figura hay un programa, denominado «vista», que reacciona a los acontecimientos interesantes y produce descripciones destinadas a los programas de dibujo. Esta

organización permite que cada algoritmo pueda utilizarse para generar varias figuras diferentes, dado que vistas diferentes pueden reaccionar de forma diferente ante el mismo conjunto de acontecimientos interesantes. En particular, las vistas de depuración que trazan el desarrollo de un algoritmo son fáciles de construir. Ésta es una aproximación de bajo nivel orientada a objetos para esta tarea, desarrollada para lenguajes anteriores a C++, con una utilización muy difundida.

Dado que la versión en Pascal de este libro fue la primera en escribirse, las versiones de los algoritmos escritas en este lenguaje son las que han producido la mayoría de las figuras —no se duplicó el trabajo detallado de interfaz de las vistas de las figuras para las implementaciones en C++ que aparecen en esta obra—. En futuras versiones será posible integrar en el código la creación de vistas con más detalle del que aparece realmente en el libro. Aunque poder trabajar desde un alto nivel será de gran ayuda para la programación orientada a objetos, hacerlos así en un lenguaje independiente a la medida será todavía un desafío importante.

El paquete que genera los gráficos se implementó en lenguaje PostScript y se escribió específicamente con el objeto de producir este libro. Además las imágenes se inspiraron en muchos de los diseños que se desarrollaron para el sistema interactivo, pero se volvieron a hacer para aprovechar las características de alta resolución disponibles en el dispositivo de fotocomposición utilizado para imprimir el libro. Este conjunto de rutinas realmente es «residente» en el dispositivo de impresión y toma como entradas las representaciones de alto nivel de las estructuras de datos. Así, la impresora combina los caracteres para formar un párrafo en un momento dado, y líneas, caracteres y sombras para formar un árbol, grafo o figura geométrica, en el instante siguiente. Por lo regular, un archivo de figura está constituido por el nombre de una vista y una pequeña cantidad de información que describe el tamaño de la figura y el estilo de los elementos de dibujo. Una vista tipo produce representaciones directas de las estructuras de datos (las permutaciones son listas de enteros, los árboles son «arrays de enlaces-padre», etc.). El software de diseño utiliza toda esta información para organizar los diferentes elementos de cada gráfico y realizar los detalles del dibujo.

En la primera edición de este libro, las figuras se dibujaron con tinta y pluma porque, en aquella época era difícil, si no imposible, producir dibujos de este tipo por medio de computadora. Ahora es difícil imaginar un procedimiento de generación de gráficos sin ayuda de la computadora. La creación de estas figuras a pluma sería una tarea gigantesca; incluso sería difícil escribir «a mano» las instrucciones gráficas de bajo nivel para la creación de las ilustraciones (recuérdese que han sido los algoritmos del libro los que han hecho la mayor parte de ese trabajo). Sin embargo, la contribución más importante de la computadora no ha sido la producción de las imágenes finales (quizás podría hacerse de otra manera) sino la producción rápida de las versiones intermedias del diseño de las figuras. La mayor parte de ellas son el producto de un lento ciclo de diseño que incluye varias docenas de versiones. Las ediciones futuras

posiblemente tengan figuras más complicadas, dado que las impresoras modernas tienen bastantes más recursos que las que existían cuando se hicieron estos diseños.

Un objetivo lejano de los informáticos de las últimas décadas ha sido el desarrollo de un «libro electrónico» que aporte la potencia de la computadora para la realización de nuevos medios de comunicación. Por una parte, la forma tradicional de este libro se puede contemplar como un paso atrás con respecto a los medios basados en computadoras; por otra se puede contemplar como un pequeño paso adelante hacia este objetivo.

Vocabulario técnico bilingüe¹

TÉRMINO ORIGINAL EN INGLÉS	TÉRMINO EMPLEADO EN ESTA OBRA	VOCABLOS ALTERNATIVOS DE USO COMÚN
abstract data structure	estructura de datos abstracta	estructura abstracta de datos
abstract data type	tipo de datos abstracto	TDA, ADT, tipo abstracto de datos
adaptive quadrature	cuadratura adaptativa	cuadratura adaptiva
additive congruential random number generator	generador de números aleatorios por congruencia aditiva	—
adjacency matrix	matriz de adyacencia	matriz de adyacencias
adjacency list	lista de adyacencia	lista de adyacencias, lista adyacente
adjacency structure	estructura de adyacencia	estructura de adyacencias
ancestor	antecesor	antepasado
arbitrary number	número arbitrario	—
array	array	arreglo, matriz, distribución, vector, tabla
articulation point	punto de articulación	—
average case analysis	análisis del caso medior	análisis del caso promedio
average running time	tiempo medio de ejecución	tiempo de ejecución promedio
B-tree	B-árbol	árbol B
balanced multiway merging	método de fusión múltiple balanceada	mezcla múltiple balanceada, intercalado múltiple balanceado
balanced tree	árbol equilibrado	árbol balanceado
biconnected component	componente biconexa	—
biconnectivity	biconectividad (en grafos)	—
binary search	búsqueda binaria	búsqueda dicotómica
binary string	cadena binaria	—
binary tree	árbol binario	—
binary search tree	árbol binario de búsqueda	árbol de búsqueda binaria
binary search	búsqueda binaria	—
bipartite graph	grafo bipartido	grafo bipartito
bitonic merging	fusión bitónica	mezcla bitónica, intercalado bitónico
bottom-up heap construction	construcción ascendente del montículo	—

¹ Ante la falta de un lenguaje estandarizado en castellano para las ciencias de la computación se ha elaborado el presente vocabulario con la traducción que hemos dado en este libro a los principales términos de la versión original en inglés, así como vocablos alternativos de uso común en España y América latina. Esta labor se verá compensada por el servicio que pueda prestar al lector. (*N. del E.*)

TÉRMINO ORIGINAL EN INGLÉS	TÉRMINO EMPLEADO EN ESTA OBRA	VOCABLOS ALTERNATIVOS DE USO COMÚN
bottom-up mergesort	método de ordenación por fusión ascendente	ordenación (clasificación) por mezcla ascendente
bottom-up parsing	análisis sintáctico ascendente	análisis gramatical ascendente
branching	ramificación	bifurcación
breadth-first search	búsqueda en amplitud	búsqueda primero en amplitud, búsqueda en anchura
brute-force		—
bubble sort	aproximación de fuerza bruta	clasificación de burbuja
Caesar cipher	método de cifrado de César	—
cipher	cifra	—
ciphertext	texto cifrado	—
circular list	lista circular	—
closure	clausura	—
clustering	fenómeno de agrupamiento	clustering
collision-resolution	resolución de colisiones	resolución de conflictos
combine and conquer	método de combina y vencerás	—
compiler	compilador	traductor
compiler-compilers	compilador de compiladores	—
complex roots of unity	raíces complejas de la unidad	computador, ordenador
computer	computadora	—
concrete data types	tipo de datos concretos	—
connected components	componentes conexas (en grafos)	grafo conectado
connected graph	grafo conexo	—
connectivity	conectividad (en grafos)	gramática independiente del contexto
context-free-grammar	gramática libre de contexto	gramática sensible al contexto
context-sensitive-grammar	gramática dependiente del contexto	—
convex hull	cerco convexo	—
cryptanalysis	criptoanálisis	—
cryptography	criptografía	—
cryptology	criptología	—
cryptosystem	sistema de cripto	—
cryptovariables	variables de cifrado	—
curve fitting	ajuste de curvas	ajuste de datos
cycling	bucle infinito	lazo infinito, ciclo infinito
Dag	DAG	—
decryption method	método de descifrado	—
depth-first search	búsqueda en profundidad	búsqueda primero en profundidad, recorrido en profundidad
deque	doble cola	deque, cola de doble extremo
deterministic algorithms	algoritmos deterministas	—

TÉRMINO ORIGINAL EN INGLÉS	TÉRMINO EMPLEADO EN ESTA OBRA	VOCABLOS ALTERNATIVOS DE USO COMÚN
digital search tree	árbol de búsqueda digital	—
directed graph	grafo dirigido	—
distribution counting	cuenta de distribuciones	—
divide and conquer	divide y vencerás	dividir para vencer, divide y reinarás
double-buffering	técnica de programación de doble buffer	manejo de dos buffers, manejo doble de buffers
double hashing	doble dispersión	dispersión doble
doubly linked list	lista doblemente enlazada	lista doblemente ligada, de doble enlace
dummy node	nodo ficticio	nodo falso
dynamic programming	programación dinámica	—
edge	arista (en grafos)	borde
encryption method	método de cifrado	método de encriptación
escape character	carácter de escape	—
escape sequence	secuencia de escape	—
exhaustive search	búsqueda exhaustiva	—
extendible hashing	método de dispersión extensible	método de dispersión ampliable
external path length	longitud del camino externo	longitud de la ruta externa
external node	nodo terminal o externo	—
external searching	búsqueda externa	—
external sorting	ordenación externa	clasificación externa
Fast Fourier transform	transformada rápida de Fourier	—
Fibonacci numbers	números de Fibonacci	—
file compression	compresión de archivos	compresión de ficheros
finite state machine	máquina de estados finitos	—
forest	bosque (en áboles)	—
fractals	fractales	—
free tree	árbol libre	—
fringe vertices	vértices del margen	—
garbage collection	recolección de basura	garbage collection
gerrymandering	manipulación del censo	—
graph	grafo	gráfica
graph isomorphism	isomorfismo de grafos	—
greatest increment method	método del mayor incremento	direccionalamiento disperso, cálculo de claves
hashing	dispersión	—
hash function	función de dispersión	—
head node	nodo cabeza	—
heap	montículo	estructura dinámica, cúmulo, montón
heap condition	condición de montículo	—
heap data structure	estructura de datos montículo	—
heapsort	método de ordenación por montículo	clasificación por montículo

TÉRMINO ORIGINAL EN INGLÉS	TÉRMINO EMPLEADO EN ESTA OBRA	VOCABLOS ALTERNATIVOS DE USO COMÚN
Huffman encoding	codificación de Huffman	—
hybrid search	búsqueda híbrida	—
implement	implementar	poner en práctica, implantar, representar
indexed sequential access	acceso secuencial indexado	acceso secuencial indizado
indexed sequential file	archivo secuencial indexado	archivo secuencial indizado
indirect binary search tree	árbol binario de búsqueda indirecta	—
indirect heap	montículo indirecto	—
infix	infijo	—
inheritance	herencia	—
inner loop	bucle interno	lazo interno, ciclo interno
inorder	en orden	orden simétrico
insertion sort	método de ordenación por inserción	clasificación por inserción
internal node	nodo interno	—
internal path length	longitud del camino interno	longitud de la ruta interna
interpolation search	método de búsqueda por interpolación	—
key	clave	llave, tecla
knapsack problem	problema de la mochila	—
lazy deletion	eliminación perezosa	eliminación postergada o diferida
leaf page	página hoja	—
least-squares data fitting	método de ajuste de datos por mínimos cuadrados	—
least common ancestor	antecesor común mínimo	ascendente común mínimo
left child	hijo izquierdo	—
left recursion	recursión izquierda	recursión por la izquierda
linear list	lista lineal	—
linear probing	método de exploración lineal	—
linear programming	programación lineal	—
link	enlace	ligadura
linked list	lista enlazada	lista ligada
list mergesort	método de ordenación por fusión de listas	clasificación por fusión (mezcla) de listas
master index	índice maestro	—
match	concordancia	pareo
maxflow-mincut theorem	teorema del flujo máximo y corte mínimo	—
mazes	laberintos	—
median-of-three partitioning	método de partición por la mediana de tres	—
mergesort	método de ordenación por fusión	clasificación por fusión, por mezcla, por intercalación

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO EMPLEADO EN ESTA OBRA</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
merging	fusión	mezcla, intercalación
minimum spanning tree	árbol de expansión mínimo	árbol abarcador mínimo
multiple searches	búsquedas múltiples	—
multiway merging	fusión multivía	mezcla multivía, intercalación múltiple
multiway radix searching	búsqueda por residuos múltiple	—
nearest-neighbor	vecino mas próximo	—
network flow	flujo de red	—
NP-complete problem	problema NP-completo	—
NP-completeness	compleción-NP	lo completo NP, completitud
object oriented	orientado a objetos	orientado al objeto
objective function	función objetivo	—
odd-even merge	método de fusión par-impar	mezcla (intercalación) par-impar
one-time pad	clave para una vez	—
O-notation	notación O	notación O mayúscula
open-addressing	direcciónamiento abierto	—
optimal binary search trees	árboles binarios de búsqueda óptima	—
ordered hashing	método de dispersión ordenada	—
ordered list	lista ordenada	—
oriented graph	grafo orientado	—
parallel algorithms	algoritmos paralelos	—
parse generator	generador de analizadores sintácticos	—
parse tree	árbol de análisis sintáctico	árbol de análisis gramatical
parsing	análisis sintáctico	análisis gramatical
path compression	compresión de caminos	—
Patricia	método Patricia	—
pattern matching	reconocimiento de patrones	concordancia (pareo) de patrones, modelos
perfect shuffle	mezcla perfecta	—
picture processing	procesamiento de imágenes	—
plaintext	texto en claro	—
planarity	planaridad	—
polynomial interpolation	interpolación polinómica	—
polynomial reducibility	reducción polinómica	—
polish notation	notación polaca	notación sin paréntesis
polyphase merging	método de fusión polifásica	mezcla (intercalación) polifásica
postfix	postfijo	sufijo, polaca inversa
postorder	orden posterior	postorden
preorder	orden previo	preorden
priority queue	cola de prioridad	—
product cipher	cifra producto	—
profiling	detección de perfiles	—

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO EMPLEADO EN ESTA OBRA</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
pseudo-random numbers	números pseudo aleatorios	números seudoaleatorios
public-key cryptosystems	sistemas de cripto de claves públicas	sistemas de cifrado de claves públicas
quadratic running time	tiempo de ejecución cuadrático	—
quasi-random numbers	números cuasi aleatorios	—
queue	cola	—
Quicksort	método de ordenación Quicksort, Quicksort	ordenación rápida, clasificación rápida
radix exchange sort	método de ordenación por intercambio de residuos	clasificación por intercambio de residuos
radix searching	métodos de búsqueda por residuos	—
radix sorting	ordenación por residuos	clasificación por residuos
random number	número aleatorio	—
range searching	búsqueda por rango	—
receiver	receptor	—
record	registro	—
red-black tree	árbol rojinegro	árbol rojo y negro
regular expression	expresión regular	—
replacement selection	selección por sustitución	selección por reemplazo
reverse polish notation	notación polaca inversa	—
rooted tree	árbol enraizado	árbol orientado
RSA	sistema de cripto de claves públicas RSA	—
run-length encoding	codificación por longitud de series	—
searching	búsqueda	—
search key	clave de búsqueda	llave de búsqueda
selection sort	método de ordenación por selección	clasificación por selección
sender	emisor	remitente
sentinel	centinela	—
separate chaining	método de encadenamiento separado	—
sequential searching	método de búsqueda secuencial	método de enlazado independiente
Shellsort	método de ordenación de Shell	clasificación de Shell, Shellsort
Simplex method	método simplex	—
slack variables	variables de holgura	variables inactivas
sort-merge	ordenación-fusión	clasificación-fusión, clasificación-mezcla (intercalación)
spanning tree	árbol de expansión (de un grafo)	árbol abarcador, generador
spline	spline	línea por interpolación
spline interpolation	método de interpolación spline	spline

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO EMPLEADO EN ESTA OBRA</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
stack	pila	stack
straight radix sort	método de ordenación directa por residuos	—
string	cadena	secuencia, serie, listas
string searching	búsqueda de cadenas	—
strongly connected components	componentes fuertemente conexas	componentes estrictamente conexas
subfile	subarchivo	subfichero
symmetric order	orden simétrico	—
system programming	programación de sistemas	—
systolic arrays	arrays sistólicos	arreglos sistólicos
terminal nodes	nodos terminales	—
top-down 2-3-4 tree	árbol descendente 2-3-4	—
top-down parsing	análisis sintáctico descendente	—
topological sorting	ordenación topológica	clasificación topológica
transitive closure	clausura transitiva	cerradura transitiva
traversal	recorrido	—
tree	árbol	—
trie	método de búsqueda trie (de retrieval)	trie
two-way merging	fusión de dos vías	mezcla (intercalación) de dos vías
undirected graphs	grafos no dirigidos	—
union-find	unión-pertenencia (algoritmos de)	—
unordered list	lista no ordenada	lista sin ordenar
variable-length encoding	codificación de longitud variable	—
Vigenere cipher	cifra de Vigenere	—
virtual memory	memoria virtual	—
Voronoi diagram	diagrama de Voronoi	—
weight balancing	balanceado de peso	equilibrado ponderado
weighted external path length	longitud ponderada del camino externo	—
weighted internal path length	longitud ponderada del camino interno	—
weighted graph	grafo ponderado	—
worst case	peor caso	—
x-node	x-nodo	nodo-x

Índice de programas

- adapt (cuadratura adaptativa), 616.
(ajuste por mínimos cuadrados), 606.
- Aleatorio (clase para enteros aleatorios en un intervalo dado), 559 (congruencia lineal), 562-563
(congruencia aditiva).
- (algoritmo simplex), 672.
- (análisis sintáctico de una expresión en postfija), 45.
- (árbol de expansión mínimo de un grafo), 496, 507-508.
- (árbol del camino más corto de un grafo), 503, 507-508.
- (árboles balanceados), 244, 247-248, 250.
- (árboles rojinegros), 244, 247-248, 250.
- bajarmontículo (restauración de un montículo de arriba hacia abajo), 166, 167, 169-170.
- burbuja (ordenación de burbuja), 112.
- buscar (buscar un elemento en un diccionario)
- árbol balanceado, 243.
 - árbol binario de búsqueda, 226.
 - árbol de búsqueda digital, 272.
 - árbol de búsqueda Patricia, 281.
 - búsqueda binaria, 218.
 - dispersión, 262.
 - secuencial, implementación por array no ordenado, 215.
 - secuencial, implementación por lista ordenada, 216-218.
- buscar (búsqueda en grafos), 424.
- buscar_caracter_de_noconcordancia (búsqueda de cadenas Boyer-Moore), 318.
- (búsqueda de cadenas de Boyer-Moore), 318-319.
- (búsqueda de cadenas de Rabin-Karp), 321.
- busquedabruna (búsqueda de cadenas por fuerza bruta), 309.
- busquedaKMP (búsqueda de cadenas Knuth-Morris-Pratt), 312, 313.
- busquedaRK (búsqueda de cadenas de Rabin-Karp), 321.
- cambiar (cambio de prioridad en una cola de prioridad), 161.
- cambioresiduos (ordenación por intercambio de residuos), 147.
- (caminos más cortos utilizando el método de Floyd), 519-520.
- ccw (comprobar la orientación de tres puntos de un plano), 382, 383-384, 401.
- (clausura transitiva utilizando el método de Warshall), 517.
- (clausura transitiva utilizando la búsqueda en profundidad), 517.
- Clavebits (clase para claves que incluyen bits), 146, 147, 153, 272, 273, 281, 283.
- Clavecadena (clase para claves como cadenas), 258.
- (codificación de Huffman), 359.
- Cola (clase cola), 35, 52-53, 335, 468.
- (componentes biconexas de un grafo), 477.
- (componentes fuertemente conexas de un grafo), 524-525.
- (concordancia del matrimonio estable), 548.
- (concordancia máxima en un grafo bipartido), 543.
- concordar (reconocimiento de patrones descritos por expresiones regulares), 333, 348.
- concordar_todos (reconocimiento de patrones descritos por expresiones regulares), 348.
- (convertir infija en postfija), 31.
- CP (clase cola de prioridad)
- implementación en array no ordenado, 161.
 - implementación en montículo, 164-165, 166, 167.
 - implementación indirecta, 175.
- (cuenta de distribuciones de ordenación), 124, 153.
- chicuadrado (prueba de la aleatoriedad por chi cuadrado), 564.
- (decodificación de Huffman), 361.
- dentro_rect (comprobar si un punto está dentro del rectángulo), 409, 413, 419.
- Dicc (clase diccionario)
- árbol balanceado, 244, 247, 250.
 - árbol binario de búsqueda, 226, 227.

- árbol de búsqueda digital, 272-273.
 árbol de búsqueda Patricia, 281, 283.
 búsqueda binaria, 219.
 dispersión, 261-262.
 secuencial, implementación por array no ordenado, 215.
 secuencial, implementación por lista ordenada, 216-218.
directaresiduos (ordenación directa por residuos), 153.
dispersión (cálculo de función de dispersión), 258.
 (dispersión por encadenamiento separado), 258.
 (doble dispersión), 264.
eliminacion (fase de eliminación descendente de la eliminación de Gauss), 590.
envolver (búsqueda del cerco convexo con envolventes), 396-397.
EQ (clase de clases de equivalencia), 481-483, 485.
estrella (dibujo de una estrella fractal), 63-64.
eval (evaluación de una spline), 602.
eval (evaluación polinómica para una FFT), 646.
 (evaluación de una expresión postfija), 30.
explgraham (búsqueda del cerco convexo con la exploración de Graham), 400.
expresion (análisis descendente), 341, 346.
factor (análisis descendente), 342, 347.
factorial (cálculo de la función factorial), 56.
fibonacci (cálculo de los números de Fibonacci), 56, 57.
 (flujo máximo de una red), 534-536.
fusion (fusión de dos listas enlazadas ordenadas), 182.
 (fusión de dos arrays ordenados), 180.
insersion (ordenación por inserción), 107-108, 111, 116-117.
insertar (insertar un elemento en un diccionario)
 árbol balanceado, 244.
 árbol binario de búsqueda, 227.
 árbol de búsqueda digital, 273.
 árbol de búsqueda Patricia, 283.
 búsqueda binaria, 218.
 dispersión, 262.
 secuencial, implementación por array no ordenado, 215.
 secuencial, implementación por lista ordenada, 217.
insertar (insertar un nuevo elemento en una cola
 de prioridad), 162 (lista), 168 (montículo), 175-176 (indirecto).
insitu (permutación de un array insitu), 117-118.
intercambio (intercambio de dos elementos de un array), 106.
interior (comprobar si un punto está dentro del polígono), 387.
intersec (comprobar la intersección de segmentos), 383, 387.
intersecciones (cuenta de intersecciones en un conjunto de segmentos), 429.
intrect (cuadratura, regla del rectángulo), 612.
intsimp (cuadratura, regla de Simpson), 615.
intrtrap (cuadratura, regla del trapecio), 613.
kruskal (cálculo del árbol de expansión mínimo utilizando el algoritmo de Kruskal), 500.
listaady (construcción de una estructura de adyacencia para un grafo), 457.
matrizady (construcción de una matriz de adyacencia para un grafo), 456.
mcd (encontrar el mcd de dos enteros), 10, 14.
meter (en una pila), 30-31, 32, 45-46, 49-50, 67-70, 134-135, 334, 465, 524.
 (método de árbol bidimensional para la búsqueda por rango), 417-419.
 (método de la rejilla para la búsqueda por rango), 412.
mult (multiplicación, evitando el desbordamiento), 559-560.
mult (multiplicación polinómica), 557.
 (multiplicación de matrices), 581.
obtener (de una cola), 34, 53, 468.
ordenar3 (ordenación de tres elementos), 105, 626.
ordenfusion (ordenación no recursiva basada en la función, lista enlazada), 185.
ordenfusion (ordenación recursiva basada en la fusión, array), 182.
ordenfusion (ordenación recursiva basada en la fusión, lista enlazada), 184.
ordenmonticulo (ordenación de un montículo), 168, 169.
ordenrapido (Quicksort), 128, 130 (recursiva), 134 (no recursiva).
ordenshell (ordenación de Shell), 121.
 (par más cercano, fusión bidimensional), 439.
pertenencia (comprobación de equivalencia), 482, 483, 485-486, 500.
Pila (clase pila), 29 (implementación por array), 32-33 (implementación por lista enlazada).

- pivote** (pivotado para el simplex y la eliminación de Gauss), 672.
poner (en una cola), 34, 52-53, 334, 468.
(primos utilizando la criba de Eratóstenes), 18.
(problema de la mochila, solución de programación dinámica), 650.
(problema del árbol binario de búsqueda óptima, solución de programación dinámica), 657.
(problema del producto de matrices en cadena, solución de programación dinámica), 654.
Rango (clase de búsqueda por rango)
áboles bidimensionales, 417-419.
bidimensional, método de la rejilla, 412.
unidimensional, 408-409.
recorrer (recorrido de un árbol), 49, 52, 64-70.
regla (dibujo de una regla), 59, 61-62.
sacar (de una pila), 30-31, 32, 45-46, 49, 67-70, 134-135, 334, 465.
selecc (búsqueda del k -ésimo elemento más pequeño), 140 (recursiva), 141-142 (no recursiva).
seleccion (ordenación por selección), 107.
(simulación del problema de Josephus por listas enlazadas), 23.
(solución de un sistema tridiagonal ecuaciones simultáneas), 593.
spline (cálculo de una spline cúbica), 601.
subirmontículo (restauración de un montículo de abajo hacia arriba), 164-165.
sumar (suma polinómica), 571 (arra), 571-572 (lista enlazada).
suprimir (extraer un elemento de un diccionario), 234, 430 (árbol de búsqueda binaria).
suprimir (supresión del mayor elemento de una cola de prioridad), 161 (lista), 165-166 (montículo), 175 (indirecto).
sustitución (fase de sustitución ascendente de la eliminación de Gauss), 591.
sustituir (sustitución del mayor elemento de una cola de prioridad), 167.
termino (análisis descendente), 341, 347.
theta (cálculo del pseudoángulo), 385, 396, 400.
vacio (comprobación de si una estructura de datos está vacía), 33 (pila), 35 (cola), 175-176 (cola de prioridad).
visitar (visitar vértices de un grafo, búsquedas en áboles),
búsqueda en amplitud, estructura de adyacencia, 468.
búsqueda en primera prioridad, listas de adyacencia, 495.
búsqueda en primera prioridad, matriz de adyacencia, 507.
búsqueda en profundidad de componentes fuertemente conexas, 468.
búsqueda en profundidad de puntos de articulación, 468.
búsqueda en profundidad no recursiva, estructura de adyacencia, 465.
búsqueda en profundidad recursiva, listas de adyacencia, 461.
búsqueda en profundidad recursiva, matriz de adyacencia, 463-464.
búsqueda exhaustiva, 680.
métodos de recorrido de áboles, 64-70.

Índice analítico

- 2-nodo, 238.
3-nodo, 238.
4-nodo, 238.
- ábaco, 692.
acceso secuencial indexado, 289-290.
Adleman, L., 371, 375.
agrupamiento, 264, 265.
Aho, A. V., 98, 375.
ajuste de curvas, 597-607.
ajuste de datos por mínimos cuadrados, 603-606.
algoritmo, 3.
algoritmo de barrido de segmentos (para intersección de segmentos), 424-430.
algoritmo de búsqueda de cadenas de Boyer-Moore, 316-320.
algoritmo de búsqueda de cadenas de Rabin-Karp, 320-322.
algoritmo de Dijkstra (para encontrar el camino más corto), 498.
algoritmo de Euclides (para encontrar el mcd), 10-12, 15, 373.
algoritmo de Floyd (para encontrar el camino más corto), 519-521, 649.
algoritmo de Kruskal (para encontrar el árbol de expansión mínimo), 499-502, 508-509
algoritmo de Prim (para encontrar el árbol de expansión mínimo), 498.
algoritmo de Warshall (para encontrar la clausura transitiva de un grafo), 649.
algoritmos de aproximación, 686-688.
algoritmos del cerco convexo, 391-405.
dimensión k, 397.
eliminación interior, 402, 404.
envolventes, 394-397, 404.
exploración de Graham, 397-405.
algoritmos descendentes; *ver* no recursivo.
algoritmos descendentes; *ver* programas recursivos.
algoritmos deterministas, 692.
algoritmos geométricos, 379-447, 508-510.
- búsqueda por rango, 407-421.
cerco convexo, 391-405.
elementales, 379-390.
intersección, 423-434.
problemas del punto más cercano, 435-446.
algoritmos paralelos, 623-636.
algoritmos sobre grafos, 451-551.
algoritmo de Dijkstra, 498.
algoritmo de Kruskal, 499-502.
algoritmo de Prim, 498.
árbol de expansión del camino más corto, 496, 507-508.
árbol de expansión mínimo, 492-502, 506-510.
búsqueda en amplitud, 468-472.
búsqueda en primera prioridad, 494-498.
búsqueda en profundidad, no recursiva, 464-468.
búsqueda en profundidad recursiva, 460-464.
búsqueda en profundidad, recursiva, 523.
caminos mas cortos, 502-510.
componentes conexas, 476.
componentes fuertemente conexas, 523-526.
concordancia, 539-550.
elementales, 451-474.
problema del matrimonio estable, 544-548.
todos los caminos mas cortos, 519-521.
unión-pertenencia, 479-488.
análisis aproximado, 82-83.
análisis asintótico, 82-83.
análisis de algoritmos, 5, 73-87.
análisis del caso medio, 73, 81.
análisis del peor caso, 73, 78-81.
análisis empírico de algoritmos, 91-92.
análisis numérico, 13.
análisis sintáctico, 337-350.
ascendente, 344-345.
descendente, 341-344.
desplazamiento reducción, 344-345.
recursivo descendente, 341-344.
análisis sintáctico desplazamiento-reducción, 344-345.

- árbol completo, 162-163.
árbol de expansión, 453.
árbol de expansión del camino más corto.
 comparación de métodos, 508.
 grafos densos, 507.
 grafos dispersos, 496.
árbol de expansión mínimo, 492-502, 506-510, 686-688.
 comparación de métodos, 508.
 grafos densos, 508.
 grafos dispersos, 496.
árbol de expansión mínimo euclíadiano, 508.
árbol de frecuencias de Huffman, 357-359.
árboles 2-3, 252.
árboles 2-3-4, 238, 252, 291.
árboles, 39-54, 453.
 altura, 41, 44.
 análisis sintáctico, 44-47.
 binarios, 41-42.
 completos, 162-163.
 definiciones, 39-42
 definiciones recursivas, 42.
 enraizados, 43.
 longitud del camino externo, 41, 44.
 longitud del camino interno, 41, 44, 229, 243, 484.
 longitud ponderada del camino externo, 359.
 longitud ponderada del camino interno, 657.
 ordenados, 41.
 orientados, 43.
 partición, 137.
 propiedades, 43-44.
 recorrido; *ver* recorrido de un árbol.
 representación de enlace padre, 483-486, 496.
árboles AVL, 252.
árboles B, 252, 290-294.
árboles balanceados, 237-253, 409.
árboles bidimensionales (2D), 414-421.
árboles binarios, 41-42.
 completos, 42.
 llenos, 42, 44.
 propiedades, 43-44.
árboles binarios de búsqueda, 223-234, 268-269, 425-430.
 óptima, 656-659.
 representación estándar, 223-225.
 representación indirecta, 233-234.
 representación por array, 233-234.
árboles binarios de búsqueda indirecta, 233-234, 426-429.
árboles de análisis sintáctico, 44-47, 339.
árboles de búsqueda digital, 272-274.
árboles descendentes 2-3-4, 237-242.
árboles *k*-dimensionales (*kD*), 420-421.
árboles rojinegros, 242-252.
árboles transversales, 515.
aristas hacia abajo, 515.
aristas hacia arriba, 515.
aritmética, 569-583.
 grandes enteros, 579-580.
 polinómica, 570-579.
aritmética de matrices, 581-582.
arqueología, 411.
arrays, 17-19, 573.
arrays paralelos, 25, 234.
arrays sistólicos, 631-635.
asignación de memoria, 25-28.

bases de datos, 288, 407.
Batcher, K. E., 629.
Bayer, R., 291.
Bentley, J. L., 98, 447.
Betterling, W. T., 620.
biconectividad en grafos, 477-479.
bits, 146-147, 153, 272, 273, 281, 283, 361.
Bland, R.G., 671.
Borodin, A., 620.
bosque, 41, 453, 480.
bosque de búsqueda en profundidad, 463, 464.
branch and bound, 683.
Brown, M. R., 209.
bucle infinito, 670.
bucle interno, 93, 109, 131.
búsqueda, 213-303.
 árbol binario, 223-234.
 autoorganizada, 218.
 binaria, 218-220.
 externa, 287-303.
 híbrida, 279.
 interpolación, 222.
 residuos, 271-285.
 secuencial, 215-218.
búsqueda autoorganizada, 218.
búsqueda binaria, 218-223.
búsqueda de cadenas, 307-323.
 algoritmo de Boyer-Moore, 316-320.
 algoritmo de Knuth-Morris-Pratt, 311-316.
 algoritmo de Rabin-Karp, 320-322.

- búsquedas múltiples, 322.
fuerza bruta, 309-311.
heurística de no concordancia, 317-318.
búsqueda de cadenas de Knuth-Morris-Pratt, 311-316.
búsqueda en amplitud
 implementación de búsqueda en primera prioridad, 497.
 implementación de cola, 468-472.
 camino mas corto, 503.
búsqueda en disco, 287.
búsqueda en primera prioridad, 494-498, 503-504, 535-537.
búsqueda en profundidad.
 grafo dirigido, 514-519.
 implementación de búsqueda en primera prioridad, 497.
 no recursiva, 464-468.
 recursiva, 460-464, 523.
búsqueda exhaustiva, 677-689.
búsqueda externa, 287-303.
 acceso secuencial indexado, 289-290.
 árboles B, 290-294.
 dispersión extensible, 295-301.
búsqueda híbrida, 279.
búsqueda por árbol binario, 223-234.
búsqueda por interpolación, 222.
búsqueda por rango, 407-421.
 aplicación a la intersección de segmentos, 430, 433.
 árboles bidimensionales (2D), 414-421.
 árboles k -dimensionales, 420-421.
 búsqueda por rango multidimensional, 419-421.
 búsqueda secuencial, 409.
 método de la rejilla, 411-414, 420.
 preprocesamiento, 407-408.
 proyección, 410.
búsqueda por rango unidimensional, 419-421.
búsqueda por residuos, 271-285.
 árboles de búsqueda digital, 272-274.
 búsqueda por residuos múltiple, 278-280.
 claves iguales, 272.
 Patricia, 280-285.
 trie, 275-280.
búsqueda por residuos múltiple, 278-280.
búsqueda secuencial, 409.
- cadenas, 307.
cálculo de la complejidad, 78-81.
- cálculo del módulo, 580.
camino cerrado simple, 384-386.
camino más largo, 691.
caminos más cortos, 498, 502-510, 691.
Carroll, L., 39.
casi lineal, 488.
centinela, 111, 120, 131, 148, 165, 180, 181, 182, 216, 322, 382, 395, 400.
cerco convexo, 391.
cifra de Vigenere, 368.
cifra de Vernam, 369.
cifrado, 366-374.
cifrado de César, 367-369.
clase búsqueda por rango, 408, 412, 417-418.
clase clases de equivalencia, 481, 485.
clase cola, 35.
clase cola de prioridad, 161, 164-168.
clase de números aleatorios, 559, 562-563.
clase de pilas, 28-33.
clase diccionario, 215-218, 225, 244, 259, 261, 272-273.
clases de equivalencia, 480.
clases.
 búsqueda por rango, 408, 412, 417-418.
 clases de equivalencia, 481, 485.
 cola, 35, 52, 335, 468.
 cola de prioridad, 161, 164-168.
 diccionario, 215-218, 225, 244, 259, 261, 272-273.
 número aleatorio, 559, 562-563.
 pila, 28-33, 45, 49, 67-70, 134-135, 332, 465, 524.
 clausura, 326, 330.
 clausura transitiva, 515-519.
 claves, 104, 213.
 claves duplicadas; *ver* claves iguales.
 claves iguales, 214, 220, 272.
 codificación de Huffman, 357-363.
 codificación de longitud variable, 355-356.
 codificación por longitud de series, 352-355.
 cola, 33-35, 332-335, 468, 497.
 cola de prioridad, 159-177, 494-502, 506-510.
 cola de prioridad indirecta, 173-176, 496.
 combina y vencerás, 62, 185.
 Comer, D., 303.
 comparar intercambiar, 106, 625-631.
 compilador, 337, 345-348.
 compilador de compiladores, 349-350.
 compleción NP, 694-696.
 componentes conexas, 476-477.

- componentes fuertemente conexas, 523-526.
compresión de archivos, 351-363.
codificación de Huffman, 357-363.
codificación de longitud variable, 355-356.
decodificación de Huffman, 362-363.
comprobación de ciclos en grafos, 464.
concatenación, 326, 329.
concordancia, 539-550.
concordancia máxima, 539-541.
conectividad en grafos, 475-489.
conjuntos, 480.
Conway, L. C., 700.
Cook, S. A., 308, 696.
criba de Eratóstenes, 18, 36.
criptoanálisis, 365-367.
criptografía, 365-374, 557.
criptología, 365-374.
cuadratura adaptativa, 616-618.
cuadratura; *ver* integración.
cuenta de distribuciones, 124-126, 153, 156.
- Dags, 521-523.
decodificación de Huffman, 362-363.
Deo, N., 700.
deque (cola de doble extremo), 332-335.
descifrado, 367-374.
descomposición, 373.
detección de perfiles, 75.
diagrama de Voronoi, 435, 443-445, 509.
diccionario de excepciones, 268.
Dijkstra, E., 498, 507, 551.
direcciónamiento abierto, 261-267.
dispersión, 255-270, 300, 320-322.
 agrupamiento, 264, 265.
 direcciónamiento abierto, 261-267.
 doble dispersión, 264-267.
 encadenamiento separado, 258-261.
 exploración lineal, 261-263.
 métodos avanzados, 267-269.
 supresión, 267.
dispersión extensible, 294-301.
dispersión ordenada, 268.
distribución de claves, 370.
divide y vencerás, 58-64, 128, 436, 576-579, 581-582, 640-642.
división, 238-241, 244-250, 292, 294, 294-300.
doble buffer, 202.
doble dispersión, 264-267.
dual de Voronoi; *ver* triangulación de Delaunay.
- Edelsbrunner, H., 447.
Edmonds, J., 534.
eliminación de la recursión, 66-70, 94, 134-137.
eliminación de la recursión final, 67, 135.
eliminación descendente, 588-591.
eliminación gaussiana, 585-595, 606.
eliminación interior, 402, 404.
eliminación perezosa, 233.
eliminación.
 en árboles binarios de búsqueda, 231-233.
 en grafos, 459.
 en tablas de dispersión, 268.
encadenamiento separado, 258-261.
entrada/salida, 13.
envolventes, 394-397, 404.
Eratóstenes, criba de, 18, 36.
estabilidad en la ordenación, 105, 190, 192.
estilo de programación, 95.
estructura de adyacencia, 457-460.
estructuras de datos, 3.
 abstracta, 17, 35-37, 160, 168.
 árbol 2-3-4, 238, 252, 291.
 árbol B, 252, 290-294.
 árbol binario de búsqueda, 223-234, 268-269, 425-430.
 árbol rojinegro, 242-252.
 árboles descendentes 2-3-4, 237-242.
 array, 17-19, 573.
 cadena, 307.
 cola, 33-35, 332-335, 468, 497.
 cola de doble extremo (deque), 332-335.
 cola de prioridad, 159-177, 494-502, 506-510.
 estructura de adyacencia, 457-460.
 lista no ordenada, 161, 215.
 lista circular, 24.
 lista doblemente enlazada, 24.
 lista enlazada, 20-25, 571-572.
 lista ordenada, 162, 215.
 listas de adyacencia, 457-460.
 matriz de adyacencia, 455-456.
 montículo, 162-176, 199-202, 359-360.
 pila, 28-33, 45, 49, 67-70, 134-135, 332-334, 465, 498.
evaluación de expresiones aritméticas (postfija), 30.
examen por anticipado, 342.
exploración de Graham, 397-405.
exploración lineal, 261-263.
exponenciación, 371-373, 574, 580.
expresión regular, 396.

- factores constantes, 79-80.
 factorial, 56.
 Fagin, R., 294, 303.
 FIFO (primero en entrar, primero en salir), 34, 53.
 filtros, 106.
 Flannery, B. P., 620.
 Floyd, R., 519.
 flujo de red, 529-538.
 flujos generalizados, 536.
 Ford, L. R., 531.
 fórmula de interpolación de Lagrange, 575, 638-639.
 Forsythe, G. E., 620.
 fractales, 64.
 Fredman, M. L., 209.
 Friedman, J. H., 447.
 fuente, 531.
 fuerza bruta, 90.
 Fulkerson, D. R., 531.
 función objetivo, 662.
 función polifásica, 204-206.
 funciones de dispersión, 256-258.
 funciones virtuales, xvi, 37.
 fusión, 180-182, 628-631.
 fusión bitónica, 629.
 fusión dividir e intercalar, 628-631.
 fusión hasta el vaciado, 204-206.
 fusión múltiple, 197-199.
 fusión múltiple balanceada, 197-199.
 fusión par-impar, 629.
- Garey, M. R., 700.
 generador de analizadores sintácticos, 349.
 generador de números aleatorios por congruencia aditiva, 561-563.
 generador de números aleatorios por congruencia lineal, 557-561.
 generadores de números aleatorios.
 combinación, 566.
 congruencia aditiva, 561-563.
 congruencia lineal, 557-561.
 mal, 560, 565.
 geometría Manhattan, 424.
 Golin, M. J., 447.
 Gonnet, G. H., 98, 209, 303.
 Gosper, R. W., 309.
 goto, 111,
 grafo, 452.
 árbol de expansión, 453.
- arista, 452.
 bipartido, 541-544.
 camino, 453.
 completo, 453.
 dirigido, 454, 459.
 dirigido acíclico, 521.
 estructura de adyacencia, 457-460.
 matriz de adyacencia, 455-456.
 no dirigido, 454.
 ponderado, 454, 459.
 representación, 454-460, 500.
 vértice, 452.
- grafo completo, 453.
 grafo dirigido, 454, 459.
 grafo no dirigido, 454.
 grafos bipartidos, 541-544.
 grafos ponderados, 454, 459, 491-511, 540.
 Graham, R. L., 98, 397, 447.
 gramáticas dependientes del contexto, 340.
 gramáticas libres de contexto, 338-340.
 grandes enteros, 679-680.
 Guibas, L., 303.
- herencia, xvi.
 heurística de la no concordancia, 318.
 Hoare, C. A. R., 127, 209.
 Hoey, D., 423, 447.
 Hopcroft, J. E., 98.
 Hu, T. C., 700.
 Huffman, D. A., 356, 375.
- implementación de algoritmos, 89-99.
 infijo, 30, 45-46.
 integración, 609-619.
 integración simbólica, 610-611.
 intercambio, 106.
 intersección, 423-434.
 dos segmentos de línea, 382-384.
 segmentos horizontales y verticales, 424-430.
 segmentos en general, 431-434.
 intersección de segmentos de líneas, 382-384.
 investigación operativa, 529.
 isomorfismo de grafos, 473.
- Jarvis, R. A., 447.
 Johnson, D. S., 700.
- Kahn, D., 375.
 Karp, R. M., 309, 375, 534.

- Kernighan, B. W., 98.
 Knuth, D. E., 98, 209, 303, 308, 375, 551, 558, 563, 620.
 Konheim, A., 375.
 Kruskal, J., 499, 551.
 Kung, H. T., 631.
- laberintos, 471-472.
 Lehmer, D., 557.
 lenguaje C, 9-16.
 lenguaje C++, 9-16.
 Lewis, H. R., 700.
- LIFO (último en entrar, primero en salir), 34, 53.
 límites inferiores, 79.
 límites superiores, 75, 79.
 Lin, S., 688.
 línea, 380.
 lista circular, 24.
 lista de adyacencia; *ver* estructura de adyacencia.
 listas de preferencia, 544-548.
 listas doblemente enlazadas, 24.
 listas enlazadas, 20-25, 216-218, 258-260, 413, 457-459, 571-572.
 circular, 24.
 doblemente enlazadas, 24.
 eliminar un elemento, 21-22.
 insertar un elemento, 21.
 nodos ficticios, 20.
 listas lineales, 35-37.
 listas; *ver* listas enlazadas.
 longitud del camino.
 externo, 41, 44.
 interno, 41, 44, 229, 243, 485.
 ponderada externa, 359-360.
 ponderada interna, 657.
 longitud del camino externo, 41, 44.
 longitud del camino interno, 41, 44, 229, 243, 485.
 longitud ponderada del camino externo, 359-360.
 longitud ponderada del camino interno, 657.
- Malcomb, M. A., 620.
 manipulación del censo electoral, 379.
 máquina de estados finitos, 314, 315.
 determinista, 327.
 no determinista, 327.
 máquina de Turing, 697.
 máquinas de cifrar/descifrar, 369-370.
 Mathematica, 620.
 matrices, 19.
- inversa, 594.
 tridiagonal, 593-594, 601.
 matrices dispersas, 581, 592.
 matrices singulares, 589.
 matriz de adyacencia, 455-456.
 máximo común divisor, 10.
 McCreight, E., 291.
mcd; *ver* máximo común divisor, método de Euclides.
 Mead, C. A., 700.
 mediana, 139-143.
 Mehlhorn, K., 303, 551.
 memoria virtual, 206, 301.
 método de Horner, 257, 573, 639.
 método de la mayor pendiente, 671.
 método de la rejilla, 411-414, 419.
 método de los mínimos cuadrados, 604-606.
 método de retroceso, 402, 681-684.
 método simplex, 663-674.
 método de Simpson, 615.
 método de Strassen (para la multiplicación de matrices), 581-582, 594.
 método del mayor incremento, 671.
 método del rectángulo, 611-613.
 método del trapecio, 613-615.
 métodos de acceso, 287-302.
 mezcla perfecta, 628-631, 645-646.
 Moler, C. B., 620.
 montículo, 162-176, 199-202, 360.
 algoritmos, 147, 164-170, 176.
 condición, 148.
 indirecto, 173-176.
 Morris, J. H., 308, 375.
 Morrison, D. R., 280.
 multiplicación de grandes enteros, 580.
 multiplicación de matrices, 581-582, 632, 653-656.
 Munro, I., 620.
- Nievergelt, J., 294, 303, 700.
 nó determinismo, 327, 693.
 no recursivo (implementación de programas recursivos).
 análisis sintáctico, 45, 344-345.
 búsqueda en profundidad, 460.
 números de Fibonacci, 57.
 ordenación por fusión, 185-188.
 ordenación rápida, 134-135.
 recorrido de un árbol, 64-70.
 selección, 141-142.

- nodo cabeza (*cabeza*), 20, 26, 27, 185-187, 217, 224-226, 250, 281, 283, 417, 418.
nodos externos, 41, 226, 275.
nodos ficticios, 46-47; *ver* nodo cabeza, z.
nodos internos, 41, 226, 275.
nodos no terminales, 40.
nodos terminales, 40.
notación O, 79.
notación polaca, 30.
notación polaca inversa, 30.
NP, 693-699.
número arbitrario, 139, 555.
números aleatorios, 139, 555, 567.
números aleatorios uniformes, 556.
números cuasi-aleatorios, 556.
números de Fibonacci, 56-58.
números primos, 320-321, 372.
números pseudo-aleatorios, 556.
- operación unión en colas de prioridad, 162, 176-177.
operaciones abstractas, 75.
Oppenheim, A. V., 700.
optimización, 93-95.
optimización de un programa, 93-95, 189-191.
or, 326, 330.
ordenación, 103-209.
 burbuja, 111-112, 113, 114.
 cinta, 195-207.
 claves iguales, 131.
 colas de prioridad, 159-177.
 cuenta de distribuciones, 124-126.
 de Shell, 119-124.
 directa por residuos, 152-155.
 disco, 207.
 empaquetado, 106.
 estabilidad, 105, 190, 192.
 externa, 104, 195-208.
 grandes registros, 116-119.
 lineal, 155-157.
 indirecta, 106, 116-119.
 interna, 104.
 método elegido, 123.
 por fusión, 179-193.
 por inserción, 109-111, 114, 115-117.
 por intercambio de residuos, 147-151, 154.
 por montículo, 168-176.
 por selección, 107-109, 112, 113, 116, 140.
 propiedades de los métodos elementales, 112-116.
- puntero, 106, 116-119.
 Quicksort, 127-139.
ordenación de burbuja, 111-112, 113, 114.
ordenación de Shell, 119-124, 128.
ordenación directa por residuos, 152-155.
ordenación externa, 104.
ordenación indirecta, 106, 161-162.
ordenación interna, 104.
ordenación por fusión, 179-193, 438-440.
ordenación por fusión de listas, 184.
ordenación por inserción, 109-111, 114, 115-117, 138.
ordenación por intercambio de residuos, 147-151, 154.
ordenación por montículos, 168-176.
ordenación por punteros, 106, 118-119, 173-175.
ordenación por residuos, 145-158.
 intercambio de residuos, 147-151, 154.
 ordenación directa por residuos, 152-157.
ordenación por selección, 107-109, 112, 113, 116, 140.
ordenación topológica, 521-523.
ordenación-fusión, 196-197.
Ottmann, T., 447.
- P, 692-699.
padre (representación de bosques por listas enlazadas), 47-48, 360-362, 481-487, 496.
páginas hoja, 297.
Papadimitriou, C. H., 551, 700.
partición, 128-131, 139, 147-151.
 árbol de la, 136-137.
 por la mediana de tres, 139.
partición por la mediana de tres, 139.
Pascal, 14.
Patashnik, O., 98.
Patricia, 280-285, 322.
permutaciones, 112, 684-686.
pila, 332.
pilas, 28-31, 45, 49, 67-70, 134-135, 332-334, 465, 525.
Pippenger, N., 294, 303.
pivotado, 589-591, 668-674.
planaridad, 472.
planificación del multiprocesador, 697.
plantillas, xvi, 106.
polígono, 380.
 cerrado simple, 384-386.
 comprobar si es un punto interior, 386.

- convexo, 388, 391-405.
representación estándar, 380.
polígono convexo, 388, 391-405.
polinomios,
 aritmética, 570-579.
 evaluación, 573-575, 640-642.
 interpolación, 575-576, 598, 642-644.
 multiplicación, 570, 576-579, 637-647.
 suma de, 570-573.
polinomios dispersos, 572.
postfijo, 30, 45-46.
pozo, 531.
Pratt, V. R., 308, 375.
Preparata, F., 447.
preprocesamiento, 407-408.
Press, W. H., 620.
previsión, 203.
Prim, R. C., 498, 551.
problema de Josefo, 23-25, 36.
problema de la mochila, 650-653, 683.
problema de la partición, 697.
problema de todos los vecinos más próximos, 443.
problema del ciclo de Hamilton, 678, 691-696.
problema del matrimonio estable, 544-548.
problema del par más cercano, 436-443.
problema del producto de matrices en cadena, 653-656.
problema del vecino más próximo, 435.
problema del vendedor ambulante, 384, 472, 677, 695.
problema euclíadiano del vendedor ambulante, 686-688.
problemás del punto mas cercano, 435-446.
problemas NP-completos, 691-699.
programa lineal no factible, 665.
programación de sistemas, 95-96, 202-204, 287-288.
programación defensiva, 95.
programación dinámica, 649-660.
programación lineal en enteros, 697.
programación lineal, 529, 661-675.
programación orientada a objetos, xvi, 33
programas conductores, 11,
programas recursivos, 55-71, 191-192.
 análisis sintáctico recursivo descendente, 341-344.
 árboles de llamada, 57, 441.
 búsqueda en profundidad, 460.
 compilador recursivo descendente, 344-348.
definiciones de árbol, 41-42.
ordenación por fusión, 182-184.
problema del punto mas cercano, 440.
Quicksort, 131.
recursividad izquierda, 343.
selección, 140.
proyección, 410.
prueba de chi-cuadrado (χ^2), 562-565.
puntero cabeza, 35.
puntero cola, 35.
punto, 380.
punto de articulación, 477.

Quicksort, 127-143, 138, 143, 405.

Rabin, M. O., 309, 375.
raíces complejas de la unidad, 639-646.
recolección de basura, 27.
reconocimiento de patrones, 325-336.
reconocimiento general de patrones descritos por expresiones regulares, 333.
recorrido de un árbol en orden posterior; *ver* recorrido de un árbol.
recorrido de un árbol en orden previo; *ver* recorrido de un árbol.
recorrido de un árbol en orden; *ver* recorrido de un árbol.
recorrido de un árbol, 49-54, 408.
 eliminación de la recursión, 64-70.
 en orden, 50, 64-66.
 orden de nivel, 52.
 orden posterior, 51.
 orden previo, 49, 66-70.
 orden simétrico, 51.
 recursivo, 64-70.
recubrimiento de vértices, 697.
recurrencias de divide y vencerás, 531.
recursión izquierda, 343.
red, 531.
reducción, 694-696.
reducción de Gauss-Jordan, 591, 594.
registros, 213.
registros particulares de retroalimentación lineal, 561.
Reingold, E. M., 700.
relaciones de recurrencia, 56-58, 83-86, 133-134, 229-230, 578.
representación,
 árboles binarios, 44-47.

- árboles binarios de búsqueda, 223-226, 233-234.
 árboles como árboles binarios, 48-49.
 bosques, 47-49.
 caracteres, 13.
 colas de prioridad, 159-177.
 conjuntos, 480.
 de colas por array, 33.
 de pilas por array, 32.
 directa de listas enlazadas mediante arrays, 25-26.
 grafos, 454-460, 500.
 grafos dirigidos, 459.
 grafos ponderados, 459.
 grandes enteros, 578-581.
 máquina de estados finitos, 330.
 matrices, 581.
 montículos como arrays, 163.
 números como bits, 146.
 pilas por listas enlazadas, 29-30.
 polinomios, 570-573.
 por pixels, 353.
 representación de bosques por el hijo más a la izquierda y el hermano más a la derecha, 48.
 representación de bosques por enlaces padres, 47, 360-362, 483-487, 496.
 Ritchie, D., 98.
 Rivest, R., 209, 371, 375.
 Roberts, E., 98, 99.
 rotación, 246-250.
 RSA sistemas de cripto de claves públicas, 371-373, 580.
 satisfactibilidad, 693.
 Schafer, R. W., 700.
 secuencia de escape, 354.
 Sedgewick, R., 209, 303, 447.
 selección, 140-143.
 selección por sustitución, 199-204.
 Sethi, R., 375.
 Shamir, A., 371, 375.
 Shamos, M. I., 423, 447.
 sintaxis, 11.
 sistema de cripto, 366.
 sistemas de cripto de claves públicas, 370-373.
 sistemas de ecuaciones lineales simultáneas, 585-595, 601, 606.
 Sleator, D. D., 209.
 Spline, 598-603.
 cuadratura, 614, 616.
 interpolación, 598-603.
- Standish, T. A., 375.
 Steiglitz, K., 551, 700.
 Stone, H. S., 700.
 Strong, H. R., 294, 303.
 Stroustrup, B., 10, 98, 99.
 sucesión de incrementos, 120.
 supercomputadora, 624, 692.
 sustitución ascendente, 588-591.
- tablas de símbolos, 213.
 Tarjan, R. E., 209, 472, 487, 524, 551.
 Template, 37.
 teorema de Cook, 696-697.
 teorema del flujo máximo y coste mínimo, 533.
 término principal, 77.
 Teukolsky, S. A., 620.
 Thompson, K., 375.
 tiempo constante, 76.
 tiempo de ejecución cuadrático, 77, 78.
 tiempo de ejecución cúbico, 77.
 tiempo de ejecución exponencial, 77.
 tiempo de ejecución $\lg N$, 78.
 tiempo de ejecución lineal, 76, 77.
 tiempo de ejecución $\ln N$, 78.
 tiempo de ejecución logarítmico, 76.
 tiempo de ejecución $N \log N$, 76, 77.
 tipo de datos abstracto, 17, 35-37, 160.
 tipos de datos, 12, 35-37.
 tipos de datos concretos, 35-37.
 todos los caminos mas cortos, 519-521.
 transformada rápida de Fourier, 631, 637-647.
 triangulación de Delaunay, 443-444.
 Tries, 275-280, 356.
- Ullman, J. D., 98, 303, 375.
 unión-pertenencia, 479-488.
- Van Leeuwen, J., 551.
 variables básicas, 669.
 variables de cifrado, 369.
 variables de holgura, 667.
 vectores, 19.
 vértices.
 árbol (en búsqueda en grafos), 468.
 en árboles y bosques, 40.
 en grafos, 452.
 margen (en búsqueda en grafos), 469.
 no vistos (en búsqueda en grafos), 469.
 vértices de un árbol, 469.

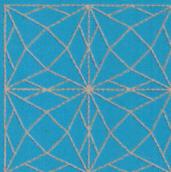
- vértices del margen, 469.
vértices no vistos, 469.
Von Neumann, J., 623.
Vuillemin, J., 209.
Warshall, S., 517.
- Wells, M. B., 700.
Wolfram, S., 620.
z, 20, 26, 27, 45-46, 50, 53, 64-70, 94, 181, 185,
188, 224-227, 244, 250, 272-274, 281, 283, 417,
418, 571-572.

<https://dogramcode.com/bloglibros>

Esta obra se terminó de imprimir en febrero del 2007
en los talleres de Ultradigital Press, S.A de C.V
Centeno 162-3, Col. Granjas Esmeralda
CP 09810, México D.F.

Algoritmos en C++

ROBERT SEDGEWICK *Princeton, University*



La última de las aportaciones de las populares series de libros de Sedgewick, conduce su amplia colección de algoritmos hacia un entorno de programación orientada a objeto (POO) con implementaciones en el lenguaje de programación C++. Estos algoritmos abarcan un amplio espectro de métodos fundamentales y avanzados: algoritmos de ordenación, de búsqueda, de procesamiento de cadenas, geométricos, sobre grafos y matemáticos. Los algoritmos están todos expresados en términos de concisas implementaciones en C++, por lo que los lectores pueden apreciar tanto sus propiedades básicas como comprobar por ellos mismos sus aplicaciones reales.

El tratamiento del análisis de los algoritmos está desarrollado cuidadosamente. Cuando se considera apropiado se presentan los resultados analíticos para ilustrar por qué se eligen ciertos algoritmos y, en algunos casos, se presentan también las relaciones entre los algoritmos prácticos y los resultados puramente teóricos.

El objetivo final de este libro es fomentar la práctica de la programación, en cualquier entorno, en cualquier lenguaje de implementación. Sedgewick describe los métodos básicos a considerar en todos los casos.

Destacan:

- Cientos de figuras detalladas que demuestran claramente cómo funcionan los algoritmos importantes.
- Por todo el libro, secciones de "propiedades" que encapsulan información específica de las características de rendimiento de los algoritmos.
- Seis capítulos de presentación de conceptos fundamentales, incluyendo una breve introducción a las estructuras de datos.

Algoritmos en C++ proporciona a los lectores las herramientas para implementar, ejecutar y depurar con seguridad los algoritmos más utilizados. Este libro es una valiosa guía para los usuarios comprometidos en la transición hacia la experimentación con la POO y/o el lenguaje C++. Se puede utilizar para autoformación o como referencia para diseñadores de sistemas de computadoras o de programas de aplicación.

SOBRE EL AUTOR:

Robert Sedgewick es profesor de Informática y Director del Departamento de Informática de la Universidad de Princeton. Ha recibido su Ph. D. por la Universidad de Stanford. El Profesor Sedgewick es una autoridad reconocida internacionalmente en el análisis de algoritmos, autor de otras versiones muy bien recibidas de este libro y editor de la *Revista de ACM, Algorítmica* y *Revista de algoritmos*.



Visítenos en:
www.pearsoneducacion.net

ISBN 968-444-401-X

9 789684 444010