Parte II

Algoritmos y fundamentos de programación

CAPÍTULO

5 Análisis de algoritmos

En la Parte I hemos visto cómo la programación orientada a objetos puede ayudar en el diseño e implementación de grandes sistemas. Ésta es, sin embargo, la mitad de la historia.

Generalmente utilizamos un ordenador porque necesitamos procesar gran cantidad de datos. Cuando ejecutamos un programa sobre una gran cantidad de datos debemos estar seguros de que el programa termina dentro de un plazo razonable. Esto es independiente, en la mayoría de los casos, del lenguaje de programación utilizado, e incluso de la metodología empleada (tales como procedimental u orientada a objetos).

Un *algoritmo* es un conjunto de instrucciones claramente especificadas que el ordenador debe seguir para resolver un problema. Una vez que se ha dado un algoritmo para resolver un problema y se ha probado que es correcto, el siguiente paso es determinar la cantidad de recursos, tales como tiempo y espacio, que el algoritmo requerirá para su aplicación. Este paso se llama *análisis de algoritmos*. Un algoritmo que necesita varios gigabytes de memoria principal no es útil en la mayoría de las máquinas actuales, aunque sea completamente correcto.

En este capítulo veremos:

- Cómo estimar el tiempo necesario para ejecutar un algoritmo.
- Diversas técnicas que reducen radicalmente el tiempo de ejecución de un algoritmo.
- Un marco matemático que describe de forma rigurosa el tiempo de ejecución de un algoritmo.
- Cómo escribir una rutina simple de *búsqueda binaria*.

5.1 ¿Qué es el análisis de algoritmos?

Más datos significa más tiempo de ejecución.

El tiempo necesario para ejecutar un algoritmo depende casi siempre de la cantidad de datos que el mismo debe procesar. Es de esperar, por ejemplo, que ordenar diez mil elementos requiera más tiempo que ordenar diez. El tiempo de ejecución de un algoritmo es, por tanto, función del tamaño de la entrada. El valor exacto de esta función depende de muchos factores, tales como la velocidad de la máquina, la calidad del compilador, y en algunos casos, de la calidad del programa. Para un programa fijo ejecutándose sobre un ordenador dado podemos dibujar la gráfica que representa la función del tiempo de ejecución. La Figura 5.1 ilustra una de estas gráficas sobre cuatro programas. Las curvas representan cuatro funciones típicas en el análisis de algoritmos: lineal, $O(N \log N)$, cuadrática y cúbica. El tamaño de la entrada N varía de 1 a 100 elementos, y los tiempos de ejecución

asociados varían de 0 a 10 milisegundos. Un simple vistazo a las Figuras 5.1 y 5.2 sugiere que las curvas lineal, $O(N \log N)$, cuadrática y cúbica representan tiempos de ejecución en orden decreciente de preferencia.

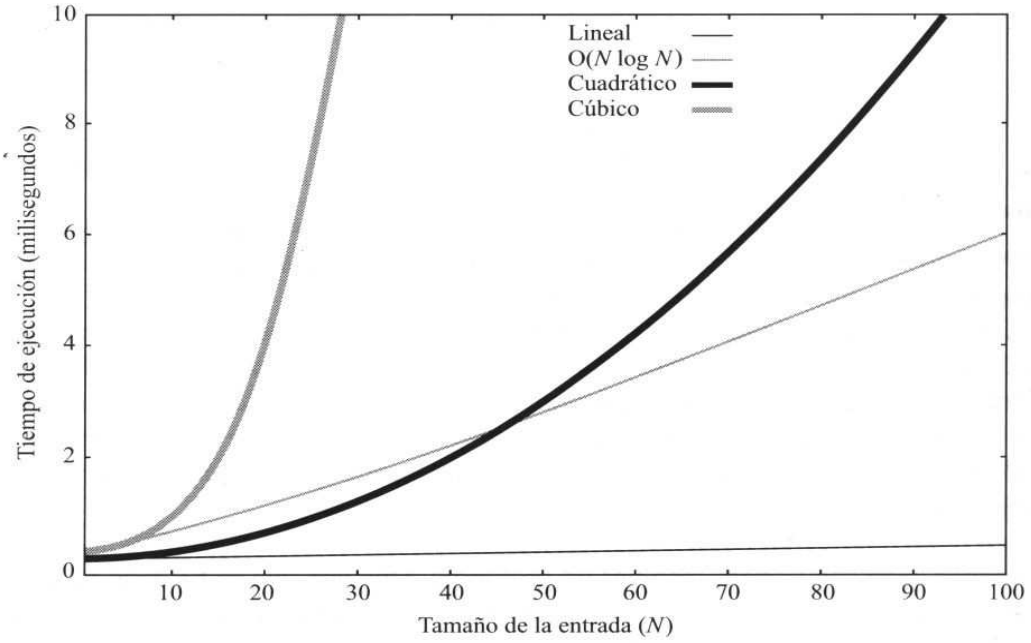

Figura 5.1 Tiempos de ejecución para tamaños de entrada pequeños.

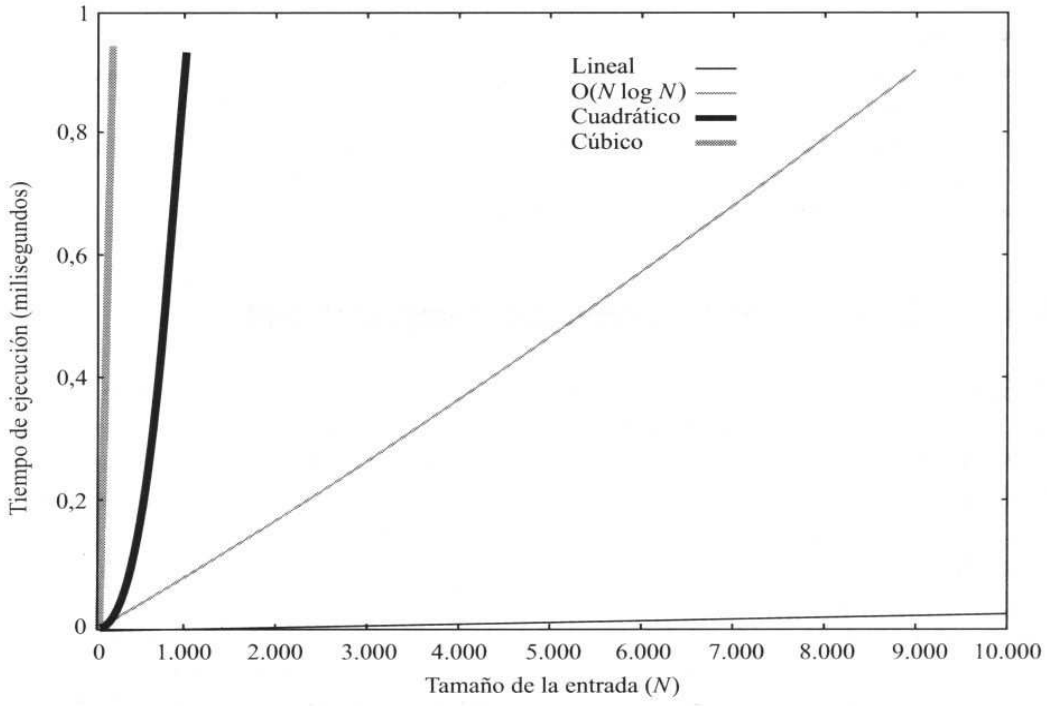

Figura 5.2 Tiempos de ejecución para tamaños de entrada mayores.

El problema de descargar un fichero de Internet nos sirve como ejemplo. Supongamos que hay un retraso inicial de 2 segundos (para establecer la conexión), después del cual la descarga se lleva a cabo a 1,6 K/seg. Entonces, si el tamaño del fichero es de N kilobytes, el tiempo de descarga viene descrito por la fórmula $T(N) = N/1{,}6 + 2$. Ésta es una *función lineal*. Podemos ver que la descarga de un fichero de 80K tardaría aproximadamente 52 segundos, mientras que descargar un fichero el doble de grande (160K) consumiría alrededor de 102 segundos, es decir casi el doble. Esta propiedad, por la cual el tiempo de ejecución es esencialmente proporcional al tamaño de la entrada, caracteriza a un *algoritmo lineal*, y es el escenario más favorable.

Como puede verse en las gráficas, algunas de las curvas no lineales conducen a tiempos de ejecución mayores. Esto indica, en particular, que un algoritmo lineal es mucho mejor que uno cúbico.

De las funciones comunes que aparecen en el análisis de algoritmos, la lineal representa el algoritmo más eficiente.

Este capítulo trata varias cuestiones importantes:

- ¿Es importante estar siempre sobre la curva más favorable?
- ¿Cuánto mejor es una curva que otra?
- ¿Cómo decidimos sobre qué curva se encuentra un algoritmo particular?
- ¿Cómo diseñamos algoritmos que eviten estar sobre las curvas malas?

Una *función cúbica* es una función cuyo término dominante es N^3 multiplicado por alguna constante. Por ejemplo, $10N^3 + N^2 + 40N + 80$ es una función cúbica. De igual manera, una función cuadrática tiene como término dominante N^2 por alguna constante, y una función lineal tiene como término dominante N por una constante. $O(N \log N)$ representa una función cuyo término dominante es N veces el logaritmo de N. El logaritmo es una función que crece lentamente; por ejemplo, el logaritmo de 1.000.000 (en base 2) es sólo 20. La función logaritmo crece más lentamente que la raíz cuadrada, la raíz cúbica, o cualquier raíz. El logaritmo se estudia en mayor profundidad en la Sección 5.5.

La tasa de crecimiento de una función se revela especialmente importante cuando N es suficientemente grande.

Dadas dos funciones, una cualquiera puede ser menor que la otra en un punto dado, por lo que en general no tiene sentido afirmar, por ejemplo, que $F(N) < G(N)$. No obstante, nosotros medimos el índice de crecimiento de las funciones. Esto viene justificado por tres razones. Primero, para funciones cúbicas, como la mostrada en la Figura 5.2, cuando N es 1.000 el valor de la función está determinado casi completamente por el término cúbico. En la función $10N^3 + N^2 + 40N + 80$, para $N = 1.000$, el valor es 10.001.040.080, del cual 10.000.000.000 es debido al término $10N^3$. Si utilizáramos solo el término cúbico para estimar el valor total de la función, obtendríamos un error de, aproximadamente, el 0,01 por ciento. Para un N suficientemente grande, el valor de la función está completamente determinado por su término dominante (el significado del término *suficientemente grande* varía para cada función).

La segunda razón por la cual medimos la tasa de crecimiento de las funciones es que el valor exacto del coeficiente del término dominante no se conserva al cambiar de máquina (aunque los valores relativos de este coeficiente para funciones de crecimiento idéntico pueden serlo). Por ejemplo, la calidad del compilador podría tener gran influencia sobre esta constante. Y tercero, los valores pequeños de N generalmente no son importantes. Para $N = 20$, la Figura 5.1 muestra que todos los algoritmos terminan antes de 5 ms. La diferencia entre el mejor y el peor algoritmo es menor que un parpadeo de ojos.

La notación O se utiliza para capturar el término más dominante de una función.

Utilizaremos la notación O para representar el índice de crecimiento. Por ejemplo, el tiempo de ejecución de un algoritmo cuadrático se describe como $O(N^2)$ (pronunciado «del orden de N al cuadrado»). La notación O nos permite establecer un orden relativo entre funciones, comparando los términos dominantes. La notación O se discute más formalmente en la Sección 5.4.

Para valores pequeños de N (por ejemplo, aquellos menores que 30), la Figura 5.1 muestra que hay puntos para los cuales una curva es inicialmente mejor que otra, aunque a la postre esto deja de ser cierto. Por ejemplo, la curva cuadrática es mejor inicialmente que la curva $O(N \log N)$, pero cuando N se hace suficientemente grande el algoritmo cuadrático pierde esta ventaja. Para valores pequeños del tamaño de la entrada, es difícil comparar las funciones porque las constantes multiplicativas llegan a ser muy significativas. La función $N + 2.500$ es mayor que N^2 cuando N es menor que 50. A partir de un punto, la función lineal será siempre menor que la función cuadrática. Lo que es más importante, para tamaños pequeños de la entrada los tiempos de ejecución son insignificantes, por lo que no tenemos que preocuparnos de ellos. Por ejemplo, la Figura 5.1 muestra que cuando N es menor que 25, todos los algoritmos descritos para este problema se ejecutan en menos de 10 ms. Por lo tanto, cuando los tamaños de la entrada son muy pequeños una regla práctica es utilizar el algoritmo más sencillo.

La Figura 5.2 muestra claramente las diferencias entre las diferentes curvas cuando el tamaño de la entrada es grande. Un algoritmo lineal resuelve un problema de tamaño 10.000 en una pequeña fracción de segundo. El algoritmo $O(N \log N)$ utiliza aproximadamente diez veces este tiempo. Nótese que las diferencias de tiempo dependen de las constantes involucradas y, por tanto, podrían ser mayores o menores. Al depender de estas constantes, un algoritmo $O(N \log N)$ podría ser más rápido que uno lineal para tamaños de la entrada bastante grandes. Sin embargo, en la práctica, para algoritmos de complejidad lógica similar, los algoritmos lineales suelen ser mejores que los algoritmos $O(N \log N)$.

Los algoritmos cuadráticos son impracticables para tamaños de la entrada que superen unos pocos miles.

Los algoritmos cúbicos son impracticables para tamaños de la entrada tan pequeños como unos pocos cientos.

Esto no es cierto, sin embargo, para los algoritmos cuadráticos y cúbicos. Los algoritmos cuadráticos son casi siempre impracticables cuando el tamaño de la entrada es mayor que unos pocos miles, y los algoritmos cúbicos son impracticables para tamaños de la entrada tan pequeños como unos pocos cientos. Para ver esto, intente ejecutar el algoritmo `ordenacionPorInsercion` de la Sección 4.6 con 100.000 elementos. Esté preparado para esperar un largo rato porque la ordenación por inserción es un algoritmo cuadrático. Los algoritmos de ordenación que se discuten en el Capítulo 8 se ejecutan en un tiempo *subcuadrático* (es decir, mejor que $O(N^2)$), lo que hace que la ordenación de grandes vectores sea practicable.

La característica más notoria de estas curvas es que los algoritmos cuadráticos y cúbicos no pueden competir con los restantes para tamaños de la entrada razonablemente grandes. Podemos implementar un algoritmo cuadrático en un lenguaje máquina altamente eficiente e implementar de forma simple un algoritmo lineal, y el algoritmo cuadrático seguirá perdiendo. Incluso los trucos de programación más inteligentes no pueden hacer que sea rápido un algoritmo ineficiente. Por tanto, antes de perder tiempo intentando optimizar un código, debemos tratar de optimizar el algoritmo. La Figura 5.3 ordena en orden creciente de tasa de crecimiento distintas funciones que describen comúnmente el tiempo de ejecución de los algoritmos.

Función	Nombre
c	Constante
$\log N$	Logarítmica
$\log^2 N$	Logarítmica al cuadrado
N	Lineal
$N \log N$	$N \log N$
N^2	Cuadrática
N^3	Cúbica
2^N	Exponencial

Figura 5.3 Funciones en orden creciente de índice de crecimiento.

5.2 Ejemplos de tiempo de ejecución de algoritmos

Esta sección examina tres problemas. También esboza posibles soluciones y determina qué clase de tiempo de ejecución tendrán los correspondientes algoritmos, sin dar programas detallados. El objetivo de la sección es proporcionar al lector cierta intuición sobre el análisis de algoritmos. La próxima sección proporciona más detalles de este proceso, mientras que la Sección 5.4 da un enfoque formal al problema del análisis de algoritmos.

Los problemas que examinamos en esta sección son:

ELEMENTO MÍNIMO DE UN VECTOR
Dado un vector de N elementos, encontrar el elemento más pequeño.

PUNTOS MÁS CERCANOS EN EL PLANO
Dados N puntos en un plano (es decir, un conjunto de pares de coordenadas x-y), encontrar el par de puntos que se encuentran más cercanos.

PUNTOS COLINEALES EN EL PLANO
Dados N puntos en un plano (es decir, en un sistema de coordenadas x-y), determinar si existen tres que se encuentren en línea recta.

El problema del elemento mínimo es fundamental en computación. Puede ser resuelto de la siguiente manera:

1. Mantener una variable `min` que almacene el elemento mínimo.
2. Inicializar `min` al primer elemento.
3. Hacer un recorrido secuencial del vector, actualizando `min` de forma adecuada.

El tiempo de ejecución de este algoritmo será lineal, es decir $O(N)$, pues repetimos una cantidad fija de trabajo por cada elemento del vector. Un algoritmo lineal es todo lo bueno que podemos esperar. Esto es debido a que tenemos que

examinar cada elemento del vector, un proceso que inexorablemente requiere un tiempo lineal.

El problema del par de puntos más cercano es un problema fundamental en las artes gráficas que podemos resolver de la siguiente manera:

1. Calcular la distancia entre cada par de puntos.
2. Conservar la mínima distancia.

Sin embargo éste es un cálculo costoso, ya que hay $N(N-1)/2$ pares de puntos[1]. Por tanto, hay aproximadamente N^2 pares de puntos. Examinar todos estos pares y calcular la distancia mínima entre ellos supondrá un tiempo cuadrático. Existe un algoritmo mejorado que se ejecuta en tiempo $O(N \log N)$ que trabaja evitando el cálculo de todas las distancias. Existe además un algoritmo del que se supone que tarda un tiempo $O(N)$. Estos dos últimos algoritmos utilizan sutiles observaciones para proporcionar resultados de forma más rápida, y están más allá del alcance de este texto.

El problema de los puntos colineales es importante en muchos algoritmos gráficos. Esto se debe a que la existencia de puntos colineales introduce un caso degenerado que requiere un tratamiento especializado. Puede ser resuelto directamente enumerando todos los grupos de tres puntos. Esta solución es aún más costosa computacionalmente que la del problema del par de puntos más cercano, ya que el número de grupos diferentes de tres puntos es $N(N-1)(N-2)/6$ (utilizando un razonamiento similar al usado para el problema del par de puntos más cercano). Esto nos indica que el enfoque directo nos llevaría a un algoritmo cúbico. Existe también una estrategia más inteligente (también más allá del alcance de este texto) que resuelve el problema en un tiempo cuadrático (y mejoras adicionales del mismo son un área de continua investigación).

La siguiente sección trata un problema que ilustra la diferencia entre algoritmos lineales, cuadráticos y cúbicos. Muestra también cómo el rendimiento de estos algoritmos se compara con una predicción matemática. Por último, después de discutir las ideas básicas, se estudia la notación O de manera más formal.

5.3 El problema de la subsecuencia de suma máxima

En esta sección consideramos el siguiente problema:

PROBLEMA DE LA SUBSECUENCIA DE SUMA MÁXIMA

Dada la secuencia de enteros (posiblemente negativos) $A_1, A_2, \ldots, A_N$, encontrar (e identificar la subsecuencia correspondiente) el valor máximo de $\sum_{k=i}^{j} A_k$. Cuando todos los enteros son negativos entendemos que la subsecuencia de suma máxima es la vacía, siendo su suma cero.

Por ejemplo, para la entrada $\{-2, \mathbf{11}, \mathbf{-4}, \mathbf{13}, -5, 2\}$, la respuesta es 20. La misma corresponde a la subsecuencia que abarca los elementos del segundo al cuarto

[1] Para ver esto, nótese que cada uno de los N puntos puede ser emparejado con $N-1$ puntos, lo que nos lleva a un total de $N(N-1)$ pares. Sin embargo, así se cuentan dos veces los pares A, B y B, A, por lo que hay que dividir por dos.

(mostrados en negrita). Como un segundo ejemplo, para la entrada {1, −3, **4**, **–2**, **–1**, **6**}, la respuesta es 7, y en este caso la subsecuencia abarca los últimos cuatro elementos.

Los detalles de programación se consideran tras realizar el diseño del algoritmo.

En Java, los vectores empiezan en el índice cero, por lo que un programa en Java representaría la entrada como una secuencia desde A_0 hasta A_{N-1}. Éste es un detalle de programación por lo que no forma parte del diseño del algoritmo.

Siempre hay que considerar el caso vacío.

Antes de discutir los distintos algoritmos para este problema, merece la pena comentar el caso degenerado en el cual todos los enteros son negativos. El enunciado del problema establece una suma máxima de cero para este caso. Uno podría preguntarse por qué lo hacemos así, en vez de devolver simplemente el mayor (es decir, el menor en magnitud) entero negativo. La razón es que la subsecuencia vacía, formada por ningún entero, también es una subsecuencia, y su suma es claramente cero. Esto es análogo al hecho de que el conjunto vacío es subconjunto de cualquier conjunto. Es importante ser consciente de que el caso vacío es siempre una posibilidad y que en muchas situaciones no es un caso especial.

Hay muchos algoritmos radicalmente diferentes (en términos de eficiencia) que pueden ser utilizados para resolver el problema de la subsecuencia de suma máxima.

El problema de la subsecuencia de suma máxima es interesante, principalmente, porque hay muchos algoritmos para resolverlo; y el rendimiento de estos algoritmos varía radicalmente. Esta sección discute tres de estos algoritmos. El primero es un algoritmo inmediato de búsqueda exhaustiva. Lamentablemente, también es muy ineficiente. El segundo es una mejora sobre el primero realizada a partir de la observación de un hecho simple. El tercero es un algoritmo muy eficiente, pero nada obvio. Probaremos que su tiempo de ejecución es lineal.

El Capítulo 7 presenta un cuarto algoritmo, con un tiempo de ejecución $O(N \log N)$. Dicho algoritmo no es tan eficiente como el algoritmo lineal, pero sí bastante más eficiente que los otros dos, siendo un ejemplo típico de la clase de algoritmos con tiempo de ejecución $O(N \log N)$. Las gráficas mostradas en las Figuras 5.1 y 5.2 son representativas de estos cuatro algoritmos.

5.3.1 El algoritmo $O(N^3)$ obvio

Un algoritmo de fuerza bruta es habitualmente el método más simple de implementar pero también el menos eficiente.

El algoritmo más simple consiste en una búsqueda exhaustiva, o un algoritmo de fuerza bruta como se muestra en la Figura 5.4. Las líneas 9 y 10 controlan dos bucles que iteran sobre todas las posibles subsecuencias. Para cada posible subsecuencia, el valor de su suma se calcula en las líneas de la 12 a la 15. Si esa suma es mejor que la calculada hasta el momento, entonces se actualiza el valor de `sumaMax`, que finalmente se devuelve en la línea 25. Dos valores de tipo `int` —`secIni` y `secFin` (atributos estáticos de la clase)— se actualizan también cuando se encuentra una subsecuencia mejor.

El algoritmo de búsqueda exhaustiva tiene el mérito de ser extremadamente simple; cuanto menos complejo es un algoritmo, más probable es que se programe con corrección. Sin embargo, por lo general, los algoritmos de búsqueda exhaustiva no son tan eficientes como sería posible. El resto de esta sección muestra que el tiempo de ejecución del algoritmo es cúbico. Contaremos cuántas veces (como función del tamaño de la entrada) se evalúan las expresiones de la Figura 5.4. Sólo necesitamos un resultado en notación O, así que una vez encontrado el término dominante, podremos ignorar otros términos de menor grado y las constantes multiplicativas.

```
1  /**
2   * Algoritmo cúbico para la subsecuencia de suma máxima.
3   * secIni y secFin representan la secuencia mejor actual.
4   */
5  public static int subsecuenciaSumaMaxima( int [ ] a )
6  {
7      int sumaMax = 0;
8
9      for( int i = 0; i < a.length; i++ )
10         for( int j = i; j < a.length; j++ )
11         {
12             int sumaActual = 0;
13
14             for( int k = i; k <= j; k++ )
15                 sumaActual += a[ k ];
16
17             if( sumaActual > sumaMax )
18             {
19                 sumMax = sumaActual;
20                 secIni = i;
21                 secFin = j;
22             }
23         }
24
25     return sumaMax;
26 }
```

Figura 5.4 Algoritmo cúbico para el problema de la subsecuencia de suma máxima.

El tiempo de ejecución del algoritmo está dominado completamente por el bucle `for` más interno de las líneas 14 y 15. Cuatro expresiones se ejecutan de forma repetida:

1. La inicialización `k = i`
2. La comprobación `k <= j`
3. La modificación `sumaActual += a[ k ]`
4. El incremento `k++`

Se utiliza un razonamiento matemático para contar el número de veces que ciertas instrucciones son ejecutadas.

El número de veces que se ejecuta la instrucción 3 la hace el término dominante de entre las cuatro expresiones. Para ver que esto es cierto, observe primero que cada inicialización está acompañada de al menos una comprobación. Nosotros ignoramos las constantes, por lo que podemos descartar el coste de las inicializaciones; éstas no pueden ser el coste dominante del algoritmo. Ya que la comprobación de la expresión 2 es falsa exactamente una vez por bucle, el número de comprobaciones sin éxito realizadas por la expresión 2 es exactamente el mismo que el de inicializaciones. El número de comprobaciones con éxito, el número de modificaciones realizadas por la expresión 3 y el número de incrementos realizados por la expresión 4 son todos idénticos. Por tanto, el número de modificaciones realizadas por la expresión 3 (es decir, el número de veces que se ejecuta la línea 15) es una medida dominante del trabajo realizado por el bucle más interno.

El número de veces que se ejecuta la línea 15 es exactamente igual al número de ternas ordenadas (i, j, k) que satisfacen $1 \leqslant i \leqslant k \leqslant j \leqslant N$.[2] Esto es así porque

[2] En Java, los índices varían de 0 a $N - 1$. Hemos utilizado el equivalente algorítmico de 1 a N, para simplificar el análisis.

el índice i recorre el vector entero, j varía desde i al final del vector, y k varía desde i hasta j. Una estimación rápida y burda nos lleva a que el número de ternas es algo menor que $N \times N \times N$, o N^3, pues i, j y k pueden cada uno tomar uno de los N valores. La restricción adicional $i \leqslant k \leqslant j$ sirve para reducir este valor. Un cálculo preciso es algo difícil de obtener y se lleva a cabo en el Teorema 5.1.

La parte más importante del Teorema 5.1 no es su demostración, sino más bien el resultado. Hay dos formas de calcular el número de ternas. Una consiste en evaluar la suma $\Sigma_{i=1}^{N} \Sigma_{j=1}^{N} \Sigma_{k=i}^{j} 1$. Podríamos calcular esta suma de dentro a fuera (*véase* Ejercicio 5.8). Utilizaremos, en cambio, una forma alternativa.

Teorema 5.1

El número de ternas ordenadas de enteros (i, j, k) que satisfacen $1 \leqslant i \leqslant k \leqslant j \leqslant N$ es $N(N+1)(N+2)/6$.

Demostración

Coloque las siguientes $N+2$ bolas en una caja: N bolas numeradas desde 1 hasta N, una bola roja sin numerar, y una bola azul sin numerar. Retire tres bolas de la caja. Si ha extraído una bola roja, numérela con el menor número de las bolas extraídas. Si ha extraído una bola azul, numérela con el número más alto de las bolas extraídas. Observe que si sacamos la bola roja y la bola azul, el efecto será que tendremos tres bolas numeradas de la misma manera. Ordene las tres bolas. Cada posible ordenación corresponde a una terna solución de la ecuación en el Teorema 5.1. El número de ordenaciones posibles es el número de formas diferentes de extraer tres bolas sin reemplazamiento de una colección de $N+2$ bolas. Esto es similar al problema de seleccionar tres puntos de un conjunto de N, que calculamos en la Sección 5.2, por lo que inmediatamente obtenemos el resultado deseado.

La consecuencia del Teorema 5.1 es que el bucle `for` más interno justifica un tiempo de ejecución cúbico. El resto de trabajo del algoritmo no tiene consecuencias ya que se repite, como mucho, una vez por iteración del bucle más interno. Dicho de otra forma, el coste de las líneas 17 a 22 no tiene consecuencias porque se ejecuta solamente tan a menudo como la inicialización del bucle `for` más interno, en vez de tan a menudo como el cuerpo de este bucle. En consecuencia, el algoritmo es $O(N^3)$.

El argumento combinatorio previo nos permite obtener un cálculo preciso del número de iteraciones del bucle más interno. Para un cálculo en un orden de crecimiento O, esto no es realmente necesario; sólo necesitamos saber que el término principal es un número constante de veces N^3. Mirando el algoritmo, vemos un bucle de tamaño potencial N dentro de un bucle de tamaño potencial N dentro de otro bucle de tamaño potencial N. Esto nos dice que el triple bucle genera potencialmente $N \times N \times N$ iteraciones. Este potencial es sólo alrededor de seis veces mayor que lo que nuestro cálculo preciso mostraba que realmente ocurría. Ya que de cualquier manera las constantes se ignoran, podemos adoptar la regla general por la cual cuando tenemos bucles anidados podemos multiplicar el coste de la instrucción más interna por el tamaño de cada bucle anidado para obtener una cota superior. En la mayoría de los casos, esta cota superior no supondrá una gran

No necesitamos hacer cálculos precisos para producir una estimación O. En muchos casos, podemos utilizar la simple regla de multiplicar el tamaño de todos los bucles anidados. Tenga presente que bucles consecutivos no se multiplican.

sobreestimación[3]. Por tanto, un programa con tres bucles anidados, cada uno de los cuales recorre una gran porción de un vector, es probable que tenga un comportamiento $O(N^3)$. Obsérvese que tres bucles consecutivos (no anidados) tendrán un comportamiento lineal; es el anidamiento lo que lleva a una explosión combinatoria. Por consiguiente, para mejorar el algoritmo, necesitamos eliminar un bucle. Esto se logra en la próxima sección.

5.3.2 Un algoritmo mejorado $O(N^2)$

Cuando eliminamos un bucle anidado interior de un algoritmo, generalmente reducimos el tiempo de ejecución.

Si podemos eliminar un bucle del algoritmo, generalmente podremos reducir el tiempo de ejecución. ¿Cómo podemos eliminar un bucle? Obviamente, no siempre podemos. Sin embargo, el algoritmo anterior contiene muchos cálculos innecesarios. La ineficiencia que el algoritmo mejorado corrige puede verse observando que como quiera que se tiene $\Sigma_{k=i}^{j} A_k = A_j + \Sigma_{k=i}^{j-1} A_k$, el cálculo en el bucle `for` más interno de la Figura 5.4 es exageradamente costoso. Más concretamente, una vez que acabamos de calcular la suma de la subsecuencia que se extiende desde i hasta $j - 1$, calcular la suma de la subsecuencia que se extiende desde i hasta j no debería llevarnos mucho tiempo, pues sólo necesitamos una suma adicional. Sin embargo, el algoritmo cúbico no utiliza esta información. Si utilizamos esta observación, obtenemos el algoritmo mejorado mostrado en la Figura 5.5. Tenemos dos bucles anidados en vez de tres, de modo que el tiempo de ejecución es $O(N^2)$.

```
1  /**
2   * Algoritmo cuadrático para la subsecuencia de suma máxima.
3   * secIni y secFin representan la subsecuencia mejor actual.
4   */
5  public static int subsecuenciaSumaMaxima( int [ ] a )
6  {
7      int sumaMax = 0;
8
9      for( int i = 0; i < a.length; i++ )
10     {
11         int sumaActual = 0;
12
13         for( int j = i; j < a.length; j++ )
14         {
15             sumaActual += a[ j ];
16
17             if( sumaActual > sumaMax)
18             {
19                 sumaMax = sumaActual;
20                 secIni = i;
21                 secFin = j;
22             }
23         }
24     }
25
26     return sumaMax;
27 }
```

Figura 5.5 Algoritmo cuadrático para la obtención de la subsecuencia de suma máxima.

[3] El Ejercicio 5.15 ilustra un ejemplo en el cual la multiplicación del tamaño de los bucles lleva a sobreestimar el resultado O.

5.3.3 Un algoritmo lineal

Si eliminamos otro bucle, conseguiremos un algoritmo lineal.

El algoritmo es delicado. Utiliza una hábil observación para saltarse gran número de subsecuencias que no pueden ser la mejor.

Para pasar de un algoritmo cuadrático a otro lineal, necesitaríamos eliminar un bucle anidado más. Sin embargo, a diferencia de la reducción ilustrada en las Figuras 5.4 y 5.5, donde la eliminación de un bucle fue sencilla, no es fácil librarse de un bucle más. El problema es que el algoritmo cuadrático representa aún una búsqueda exhaustiva; es decir, estamos examinando todas las posibles subsecuencias. La única diferencia entre los algoritmos cúbico y cuadrático es que el coste de comprobar cada subsecuencia sucesiva es constante $O(1)$ en vez de lineal $O(N)$. Ya que el número de subsecuencias posibles es cuadrático, la única forma de conseguir una cota subcuadrática es encontrar un razonamiento astuto que nos permita dejar de considerar un gran número de subsecuencias, sin calcular sus sumas ni comprobar si esa suma es un nuevo máximo. Esta sección muestra como hacerlo.

Eliminamos primero un gran número de subsecuencias posibles. Denotaremos por $A_{i,j}$ la subsecuencia que abarca los elementos del i al j, siendo $S_{i,j}$ su suma.

Teorema 5.2

Sea $A_{i,j}$ una secuencia cualquiera con $S_{i,j} < 0$. Tenemos que si $q > j$, entonces $A_{i,q}$ no es la subsecuencia de suma máxima.

Demostración

La suma de los elementos de A desde i hasta q es la suma de los elementos de A desde i hasta j más la suma de los elementos desde $j+1$ hasta q. Tenemos, por tanto, $S_{i,q} = S_{i,j} + S_{j+1,q}$. Ya que $S_{i,j} < 0$, sabemos que $S_{i,q} < S_{j+1,q}$. Podemos entonces deducir que $A_{i,q}$ no es la subsecuencia de suma máxima.

La Figura 5.6 muestra una representación gráfica de las sumas generadas por i, j y q. El Teorema 5.2 demuestra que es posible evitar examinar diversas subsecuencias, incorporando una comprobación adicional: siempre que `sumaActual` sea menor que cero, podemos salir del bucle más interno de la Figura 5.5. Intuitivamente, si vemos una subsecuencia cuya suma es negativa, la misma no puede formar parte de la subsecuencia máxima, pues podemos obtener una subsecuencia mayor suprimiéndola. Esta observación por sí misma no es suficiente para reducir el tiempo de ejecución por debajo del cuadrático. Una observación similar también es cierta: todas las subsecuencias que bordean la subsecuencia máxima deben tener suma negativa o cero (de otra forma, las añadiríamos). Esto tampoco reduce el tiempo de ejecución por debajo del cuadrático. Sin embargo, una tercera observación, ilustrada en la Figura 5.7, sí que lo hace. La formalizamos en el Teorema 5.3.

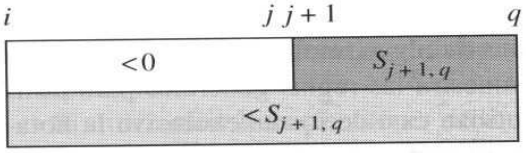

Figura 5.6 Las subsecuencias utilizadas en el Teorema 5.2.

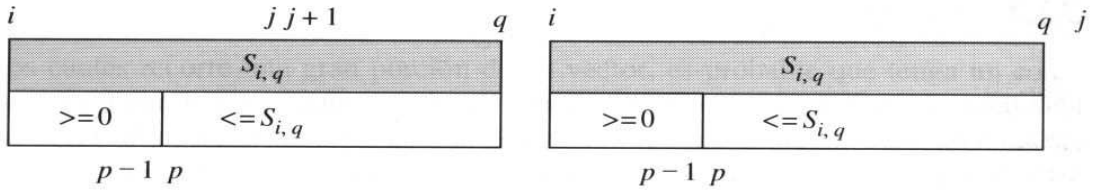

Figura 5.7 Las subsecuencias utilizadas en el Teorema 5.3. La secuencia desde p hasta q tiene una suma que es, como mucho, igual que la subsecuencia de i a q. En la parte de la izquierda la secuencia de i a q no es la máxima (por el Teorema 5.2). En la parte derecha, la secuencia de i a q ya ha sido comprobada.

Teorema 5.3

Para cada i, sea $A_{i,j}$ la primera secuencia que satisfaga $S_{i,j} < 0$. Entonces, para cualquier $i \leqslant p \leqslant j$ y $p \leqslant q$, $A_{p,q}$ o bien no es una subsecuencia máxima, o es igual a una subsecuencia máxima ya considerada.

Demostración

Si $p = i$, entonces es aplicable el Teorema 5.2. En otro caso, como en el Teorema 5.2, tenemos $S_{i,j} = S_{i,p-1}$. Puesto que j es el menor índice para el cual $S_{i,j} < 0$, se sigue que $S_{i,p-1} \geqslant 0$. Por tanto, $S_{p,q} \leqslant S_{i,q}$. Si $q > j$ (ilustrado en la parte de la izquierda de la Figura 5.7), entonces por el Teorema 5.2 $A_{i,q}$ no es una subsecuencia máxima, ni lo es $A_{p,q}$. En otro caso, como se muestra en la parte derecha de la Figura 5.7, la subsecuencia $A_{p,q}$ tiene una suma como mucho igual a la de la subsecuencia $A_{i,q}$ que ya ha sido considerada.

Si detectamos una suma negativa, podemos mover *i* tras *j*.

Si un algoritmo es complejo, es necesaria una demostración de su corrección.

El Teorema 5.3 nos dice que cuando se detecta una subsecuencia negativa, no sólo podemos salir del bucle más interno, sino que podemos avanzar `i` hasta `j+1`. La Figura 5.8 muestra cómo podemos rescribir el algoritmo utilizando un solo bucle. Claramente, el tiempo de ejecución de este algoritmo es lineal: en cada paso del bucle, avanzamos `j`, así que el bucle se itera como mucho N veces. La corrección de este algoritmo es menos obvia que en los casos anteriores. Esto es típico: los algoritmos que utilizan la estructura del problema para evitar una búsqueda exhaustiva requieren habitualmente algún tipo de demostración de corrección. Hemos probado que el algoritmo (aunque no el programa Java correspondiente) es correcto utilizando un argumento formal. Nuestro propósito no es realizar una completa discusión matemática, sino dar una idea de las técnicas que podrían ser necesarias en un trabajo más avanzado.

5.4 Reglas generales para la notación *O*

Una vez tenemos las ideas básicas del análisis de algoritmos, podemos adoptar un enfoque un poco más formal. Esta sección muestra las reglas generales para utilizar la notación O. Aunque en este texto se utiliza casi de forma exclusiva la notación O, se definen otros tres tipos de notación que están relacionados con ella y que se utilizarán de forma ocasional más adelante en el texto.

```
1  /**
2   * Algoritmo lineal para la subsecuencia de suma máxima.
3   * secIni y secFin representan la secuencia mejor actual.
4   */
5  public static int subsecuenciaSumaMaxima( int [ ] a )
6  {
7      int sumaMax = 0;
8      int sumaActual = 0;
9
10     for( int i = 0, j = 0; j < a.length; j++ )
11     {
12         sumaActual += a[ j ];
13
14         if( sumaActual > sumaMax )
15         {
16             sumaMax = sumaActual;
17             secIni = i;
18             secFin = j;
19         }
20         else if( sumaActual < 0 )
21         {
22             i = j + 1;
23             sumaActual = 0;
24         }
25     }
26
27     return sumaMax;
28 }
```

Figura 5.8 Algoritmo lineal para la obtención de la subsecuencia de suma máxima.

DEFINICIÓN: (O) $T(N)$ es $O(F(N))$ si existen constantes positivas c y N_0 tales que para $N \geqslant N_0$ se verifica $T(N) \leqslant cF(N)$.

DEFINICIÓN: (Ω) $T(N)$ es $\Omega(F(N))$ si existen constantes positivas c y N_0 tales que para $N \geqslant N_0$ se verifica $T(N) \geqslant cF(N)$.

DEFINICIÓN: (Θ) $T(N)$ es $\Theta(F(N))$ si y sólo si $T(N)$ es $O(F(N))$ y $T(N)$ es $\Omega(F(N))$.

DEFINICIÓN: (o) $T(N)$ es $o(F(N))$ si y sólo si $T(N)$ es $O(F(N))$ y $T(N)$ no es $\Theta(F(N))$.

La primera definición, correspondiente a la notación O, afirma que existe un punto N_0 tal que para todos los valores de N después de este punto, $T(N)$ está acotada por algún múltiplo de $F(N)$. Éste es el N suficientemente grande que mencionábamos anteriormente. Por tanto, si el tiempo de ejecución $T(N)$ de un algoritmo es $O(N^2)$, entonces, ignorando las constantes, podemos garantizar que a partir de un punto se puede acotar el tiempo de ejecución mediante una función cuadrática. Obsérvese que si el tiempo de ejecución real es lineal, la afirmación de que el tiempo de ejecución es $O(N^2)$ es técnicamente correcta, porque la desigualdad es válida. Sin embargo, decir que dicho tiempo es $O(N)$ sería la afirmación más precisa posible.

La notación O es similar al menor o igual, cuando consideramos tasas de crecimiento.

Si utilizamos los operadores de desigualdad tradicionales para comparar tasas de crecimiento, entonces la primera definición nos dice que la tasa de crecimiento de $T(N)$ es menor o igual que la de $F(N)$.

La notación Ω es similar al mayor o igual, cuando consideramos tasas de crecimiento.

La segunda definición, $T(N) = \Omega(F(N))$, correspondiente a la llamada *notación Omega*, nos dice que la tasa de crecimiento de $T(N)$ es mayor o igual que la de $F(N)$. Por ejemplo, podríamos decir que cualquier algoritmo que examine todas las posibles subsecuencias en el problema de la subsecuencia de suma máxima debe tener un tiempo de ejecución $\Omega(N^2)$, porque son posibles un número cuadrático de subsecuencias. Éste es un argumento sobre una cota inferior, que se utiliza en análisis más avanzados. Más tarde en este texto, veremos un ejemplo de ello, donde se demuestra que cualquier algoritmo de ordenación de propósito general requiere un tiempo $\Omega(N \log N)$.

La notación Θ es similar al igual, cuando consideramos tasas de crecimiento.

La tercera definición, $T(N) = \Theta(F(N))$, correspondiente a la denominada *notación Theta*, nos dice que la obtención de la tasa de crecimiento de $T(N)$ es igual que la de $F(N)$. Por ejemplo, el algoritmo para la subsecuencia de suma máxima de la Figura 5.5 se ejecuta en un tiempo $\Theta(N^2)$. Esto significa que el tiempo de ejecución está acotado por una función cuadrática y que esta cota no puede mejorarse, pues el algoritmo también está acotado inferiormente por otra función cuadrática. Cuando utilizamos la notación Θ, estamos proporcionando no solamente una cota superior del tiempo de ejecución del algoritmo, sino que garantizamos que el análisis que nos lleva a esta cota superior es lo más ajustado posible. A pesar de la precisión adicional facilitada por la notación Θ, la notación O se utiliza más extensamente, excepto por investigadores en el campo del análisis de algoritmos.

La notación *o* es similar al menor estricto, cuando consideramos tasas de crecimiento.

La última definición, $T(N) = o(F(N))$, correspondiente a la denominada notación o, nos dice que la tasa de crecimiento de $T(N)$ es estrictamente menor que la de $F(N)$. Esto difiere de la notación O, pues O admite la posibilidad de que las tasas de crecimiento sean iguales. Por ejemplo, si el tiempo de ejecución de un algoritmo es $o(N^2)$, se garantiza que crece con una tasa menor que cuadrática (es decir, es *subcuadrático*). Por tanto, tener como cota $o(N^2)$ es mejor que tener como cota $\Theta(N^2)$. La Figura 5.9 ilustra los significados de estas cuatro definiciones.

Cuando utilice la notación *O* elimine los coeficientes, los términos no dominantes y los símbolos relacionales.

Resulta oportuno aquí comentar un par de notas de estilo. No es de buen estilo incluir constantes o términos de orden menor que el dominante dentro de la notación O. No escriba $T(N) = O(2N^2)$ ni $T(N) = O(N^2 + N)$. En ambos casos, la forma correcta sería $T(N) = O(N^2)$. Recuerde que en cualquier análisis que requiera una respuesta O, son posibles todo tipo de abreviaturas. Los términos de orden inferior, las constantes y los símbolos relacionales pueden eliminarse todos.

Expresión matemática	Índices de crecimiento relativos
$T(N) = O(F(N))$	El crecimiento de $T(N)$ es $\leq$ que el crecimiento de $F(N)$.
$T(N) = \Omega(F(N))$	El crecimiento de $T(N)$ es $\geq$ que el crecimiento de $F(N)$.
$T(N) = \Theta(F(N))$	El crecimiento de $T(N)$ es $=$ que el crecimiento de $F(N)$.
$T(N) = o(F(N))$	El crecimiento de $T(N)$ es $<$ que el crecimiento de $F(N)$.

Figura 5.9 Significado de las diferentes funciones de crecimiento.

Una vez formalizados estos conceptos matemáticos, los pondremos en relación con el análisis de algoritmos. La regla más importante es *que el tiempo de ejecución de un bucle es como mucho el tiempo de ejecución de las instrucciones dentro del bucle (incluyendo los tests) multiplicado por el número de iteraciones.* Como vimos antes, la inicialización y la comprobación de la condición del bucle es habitualmente menos dominante que las instrucciones que forman el cuerpo del bucle.

El tiempo de ejecución de las instrucciones dentro de un grupo de bucles anidados es el tiempo de ejecución de las instrucciones (incluyendo la comprobación de la condición del bucle más interno) multiplicado por el tamaño de todos los bucles. El tiempo de ejecución de una secuencia de bucles consecutivos es igual al tiempo de ejecución del bucle dominante. La diferencia en tiempo entre dos bucles anidados cuyos índices varían de 1 a N y dos bucles consecutivos no anidados pero que se ejecutan sobre los mismos índices es la misma que la diferencia de espacio entre una matriz bidimensional y dos vectores unidimensionales. El segundo caso es lineal, pues $N+N$ es $2N$, que es $O(N)$. En ocasiones, esta simple regla puede sobrestimar el tiempo de ejecución, pero en la mayoría de los casos no lo hace. En cualquier caso no hay problemas, pues la notación O no garantiza una respuesta asintótica exacta, sino tan sólo una cota superior.

Una *cota en el caso peor* es una garantía sobre todas las entradas de cierto tamaño.

Los análisis que hemos realizado hasta ahora son *cotas en el caso peor*, que garantizan el tiempo sobre todas las entradas de cierto tamaño. Otro tipo de análisis es el *análisis en el caso medio*. En este caso, el tiempo de ejecución se obtiene como la media sobre todas las posibles entradas de tamaño N. La media podría diferir del caso peor si, por ejemplo, una instrucción condicional que depende de la entrada particular hace que el bucle termine rápidamente. Las cotas en el caso medio se discuten más formalmente en la Sección 5.8. Por ahora, tenga en cuenta que el hecho de que un algoritmo tenga mejor cota en el caso peor que otro, no implica nada sobre sus cotas en el caso medio. Sin embargo, en muchos casos las cotas en el caso peor y en el caso medio están fuertemente relacionadas. Cuando no lo estén, se discutirán por separado.

En las cotas para el caso medio, el tiempo de ejecución se obtiene como la media sobre todas las posibles entradas de tamaño N.

El último asunto referente a la notación O que discutimos es cómo crece el tiempo de ejecución para cada tipo de curva. Ya vimos esto en las gráficas de las Figuras 5.1 y 5.2. Pero queremos una repuesta más cuantitativa a la siguiente pregunta: si un algoritmo tarda un tiempo $T(N)$ en resolver un problema de tamaño N, ¿cuánto tiempo tardará en resolver un problema de tamaño mayor? Por ejemplo, ¿cuánto tardará en resolver el problema cuando el tamaño de la entrada se multiplica por diez? Las respuestas exactas experimentales se muestran en la Figura 5.10. Sin embargo, nos gustaría contestar la pregunta sin tener que ejecutar el programa, confiando en que nuestra respuesta analítica corresponda con el comportamiento observado.

Empecemos examinado el algoritmo cúbico. Por hipótesis, asumimos que el tiempo de ejecución está aproximado de manera razonable por $T(N) = cN^3$. En consecuencia tendremos $T(10N) = c(10N)^3$. Ahora unas simples manipulaciones matemáticas nos llevan a

$$T(10N) = 1.000cN^3 = 1.000T(N).$$

Por tanto, el tiempo de ejecución de un programa cúbico se incrementa por 1.000 (asumiendo un tamaño N suficientemente grande) cuando la entrada se incrementa

	Figura 5.4	Figura 5.5	Figura 7.18	Figura 5.8
N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0,00103	0,00045	0,00066	0,00034
100	0,47015	0,01112	0,00486	0,00063
1.000	448,77	1,1233	0,05843	0,00333
10.000	NA	111,13	0,68631	0,03042
100.000	NA	NA	8,01130	0,29832

Figura 5.10 Tiempos de ejecución observados (en segundos) de los diferentes algoritmos para el cálculo de la subsecuencia de suma máxima.

Si el tamaño de la entrada se incrementa por un factor *f*, el tiempo de ejecución de un algoritmo cúbico se incrementa aproximadamente por un factor f^3.

por un factor de 10. Esto se confirma de forma aproximada examinando el crecimiento en los tiempos de ejecución desde $N = 100$ hasta 1.000 en la Figura 5.10. Recuerde que no esperamos una respuesta exacta, sino una aproximación razonable. También esperaríamos que para $N = 10.000$, hubiera otro crecimiento por mil en el tiempo de ejecución.

El resultado sería que el uso de un algoritmo cúbico requeriría, más o menos, dos semanas de tiempo de cómputo. En general, si el tamaño de la entrada se incrementa por un factor f, entonces el tiempo de ejecución de un algoritmo cúbico se incrementa aproximadamente por un factor f^3.

Si el tamaño de la entrada se incrementa por un factor *f*, el tiempo de ejecución de un programa cuadrático se incrementa aproximadamente por un factor f^2.

Podemos realizar cálculos similares para los algoritmos cuadráticos y lineales. Para el algoritmo cuadrático, asumimos que $T(N) = cN^2$. Se sigue que $T(10N) = c(10N)^2$. Y al expandir, obtenemos

$$T(10N) = 100cN^2 = 100T(N).$$

Por tanto, cuando el tamaño de la entrada se incrementa por un factor 10, el tiempo de ejecución de un algoritmo cuadrático se incrementa por un factor aproximado de 100. Esto también puede confirmarse en la Figura 5.10. En general, un incremento por un factor f del tamaño de la entrada produce un incremento por un factor f^2 del tiempo de ejecución de un algoritmo cuadrático.

Si el tamaño de la entrada se incrementa por un factor *f*, entonces el tiempo de ejecución de un programa lineal también se incrementa por un factor *f*. Éste es el tiempo de ejecución más favorable para un algoritmo.

Finalmente, para un algoritmo lineal, cálculos similares mostrarían que un incremento por diez del tamaño de la entrada tiene como resultado un incremento por diez del tiempo de ejecución. Una vez más, esto puede ser confirmado experimentalmente. Observe, sin embargo, que para el programa lineal el término *suficientemente grande* significa una entrada algo mayor que en los otros casos. Esto es debido a la sobrecarga de 0,0003 segundos que aparece en todos los casos. Para un programa lineal, este término es todavía significativo para tamaños de entrada moderados.

El análisis utilizado en estos casos no sirve cuando hay términos logarítmicos. Cuando el tamaño de la entrada de un algoritmo $O(N \log N)$ se multiplica por diez, el tiempo de ejecución se incrementa en un factor que es algo mayor que

diez. De forma específica, tenemos que $T(10N) = c(10N) \log (10N)$. Cuando expandimos obtenemos

$$T(10N) = 10cN \log (10N) = 10cN \log N + 10cN \log 10 = 10T(N) + c'N.$$

donde $c' = 10c \log 10$. Cuando N se hace muy grande, el cociente $T(10N)/T(N)$ se va aproximando a 10, pues $c'N/T(N) \approx (10 \log 10)/\log N$ se va haciendo cada vez más pequeño al incrementar N. Por consiguiente, si el algoritmo es competitivo comparado con uno lineal para un N muy grande, seguirá siendo competitivo para un N un poco mayor.

¿Significa todo esto que un algoritmo cuadrático o cúbico no es útil? La respuesta es no. En algunos casos, el algoritmo conocido más eficiente es cuadrático o cúbico. En otros, el algoritmo más eficiente es incluso peor (exponencial). Además, cuando el tamaño de la entrada es pequeño, cualquier algoritmo es útil, y frecuentemente los algoritmos que no son asintóticamente eficientes son fáciles de programar. Finalmente, hay que señalar que una buena forma de evaluar un algoritmo lineal complejo es comparar su comportamiento con un algoritmo de búsqueda exhaustiva. La Sección 5.8 discute otras limitaciones del modelo O.

5.5 Logaritmos

La lista de funciones de tasas de crecimiento típicas incluye varias funciones que contienen *logaritmos*. Esta sección describe con más detalle el fundamento matemático en el que se basan los logaritmos. La siguiente sección ilustra como aparecen en un simple algoritmo.

Empezaremos con la definición formal para seguir con una serie de consecuencias.

El *logaritmo* de N (en base 2) es el valor X tal que 2 elevado a X es igual a N. Por defecto, la base en la que computaremos los algoritmos será 2.

DEFINICIÓN: Para cualquier B, $N > 0$, $\log_B N = K$ si $B^K = N$.

En esta definición, B es la base del logaritmo. En computación, cuando la base se omite, por defecto se toma el valor 2. Esto es natural por varias razones, como se verá más adelante en este mismo capítulo. Demostraremos un resultado matemático, el Teorema 5.4, en el que mostramos que, en lo que se refiere a la notación O, la base no es importante. Veremos también cómo pueden ser derivadas diversas relaciones que involucran logaritmos.

Teorema 5.4

La base no es importante. Para cualquier constante $B > 1$, $\log_B N = O(\log N)$.

Demostración

Sea $\log_B N = K$. Entonces $B^K = N$. Sea $C = \log B$. Entonces $2^C = B$. De este modo, $B^K = (2^C)^K = N$. Por tanto, tenemos que $2^{CK} = N$, lo cual implica que $\log N = CK = C \log_B N$. Por tanto, $\log_B N = (\log N)/(\log B)$, lo que completa la demostración.

En el resto del texto, utilizaremos exclusivamente logaritmos en base 2. Un hecho importante sobre los logaritmos es que crecen lentamente. Como quiera que

$2^{10} = 1.024$, $\log 1.024 = 10$. Cálculos adicionales muestran que el logaritmo de un millón es aproximadamente 20, y el logaritmo de un billón aproximadamente 40. Por consiguiente, el rendimiento de un algoritmo $O(N \log N)$ está mucho más cerca de un algoritmo lineal $O(N)$ que de uno cuadrático $O(N^2)$, incluso para tamaños de entrada moderadamente grandes. Antes de ver un algoritmo realista cuyo tiempo de ejecución incluye un logaritmo, veamos algunos ejemplos de cómo los logaritmos entran en juego.

BITS EN UN NÚMERO BINARIO

¿Cuántos bits son necesarios para representar N enteros consecutivos?

El número de bits necesarios para representar números es logarítmico.

Un entero `short` de 16 bits representa los 65.536 enteros en el rango de -32.768 a 32.767. En general, B bits son suficientes para representar 2^B enteros diferentes. Por tanto, el número de bits B necesario para representar N enteros consecutivos satisface la ecuación $2^B \geq N$. Obtenemos entonces $B \geq \log N$, de donde se sigue que el número de bits mínimo es $\lceil \log N \rceil$. (Aquí $\lceil X \rceil$ es la función techo y representa el entero más pequeño mayor o igual que X. La función suelo correspondiente, $\lfloor X \rfloor$, representa el mayor entero menor o igual que X.)

DUPLICACIONES REPETIDAS

Empezando con X = 1, ¿cuántas veces debe ser X duplicado antes de que sea mayor que N?

Empezando en 1, podemos duplicar un valor repetidamente solamente un número de veces logarítmico antes de alcanzar N.

Supongamos que empezamos con 1 euro y lo duplicamos cada año. ¿Cuánto tardaríamos en ahorrar un millón de euros? En este caso, después de un año tendríamos 2 euros; después de 2 años, 4 euros; después de 3 años, 8 euros, etc. En general, después de K años tendríamos 2^K euros, por lo que queremos encontrar el menor K que satisfaga $2^K \geq N$. Ésta es la misma ecuación de antes, por lo que $K = \lceil \log N \rceil$. Después de 20 años, tendríamos alrededor de un millón de euros. El *principio de las duplicaciones repetidas* afirma que empezando en 1, podemos duplicar sólo $\lceil \log N \rceil$ veces antes de alcanzar N.

SUCESIVAS DIVISIONES POR LA MITAD

Empezando con X = N, si N se divide de forma repetida por la mitad, ¿cuántas veces hay que dividir para hacer N menor o igual que 1?

Sólo podemos dividir por la mitad un número dado un número logarítmico de veces. Esto se utiliza para obtener rutinas de búsqueda logarítmicas.

Si la división redondea al entero más cercano (o es división real, no entera) tenemos el mismo problema que con las duplicaciones repetidas, excepto que ahora nos movemos en el sentido opuesto. Una vez más la respuesta es $\lceil \log N \rceil$ iteraciones. Si la división redondea hacia abajo, la respuesta sería $\lfloor \log N \rfloor$. La diferencia puede verse empezando con $X = 3$. Se necesitan dos divisiones, a menos que la división redondee hacia abajo, en cuyo caso sólo se necesita una.

Muchos de los algoritmos examinados en este texto tendrán logaritmos en sus tiempos de ejecución debido al *principio de sucesivas divisiones por la mitad*. Dicho principio nos dice que un algoritmo es $O(\log N)$ si tarda un tiempo constante ($O(1)$) en dividir el tamaño del problema por una fracción constante (que generalmente es 1/2). Esto se deduce directamente del hecho de que habrá $O(\log N)$

iteraciones en el bucle. Cualquier fracción constante sirve, ya que la fracción está reflejada en la base del logaritmo, y el Teorema 5.4 nos dice que la base no es importante.

Todas las apariciones de logaritmos que faltan se introducen (directa o indirectamente) aplicando el Teorema 5.5. La demostración utiliza nociones de cálculo, pero comprender la demostración no es necesario para utilizar el teorema.

El *número armónico N-ésimo* es la suma de los inversos de los *N* primeros enteros positivos. La tasa de crecimiento de los números armónicos es logarítmica.

Teorema 5.5

Siendo $H_N = \Sigma_{i=1}^{N} 1/i$ *se tiene que* $H_N = \Theta(\log N)$. *Una estimación más precisa vendría dada por* $\ln N + 0{,}577$, *donde* ln *denota el logaritmo neperiano.*

Demostración

La intuición de la demostración es que una suma discreta se puede aproximar con una integral (continua). La demostración utiliza una construcción para mostrar que la suma H_N *puede ser acotada superior e inferiormente por* $\int \frac{dx}{x}$, *con límites apropiados. Los detalles se ven en el Ejercicio* 5.17. H_N *es conocido como el número armónico N-ésimo.*

La próxima sección muestra cómo el principio de sucesivas divisiones por la mitad conduce a un algoritmo de búsqueda eficiente.

5.6 Problema de la búsqueda estática

Un uso importante de las computadoras es la búsqueda de datos. Si los datos no pueden cambiar (por ejemplo, si están almacenados es un CD-ROM), decimos que son estáticos. Una *búsqueda estática* accede a datos estáticos. El problema de la búsqueda estática se formula como sigue:

PROBLEMA DE LA BÚSQUEDA ESTÁTICA
Dado un entero X y un vector A, devolver la posición de X en A o una indicación de que no está presente. Si X aparece más de una vez, devolver cualquier aparición. El vector A nunca será modificado.

Un ejemplo de búsqueda estática es buscar a una persona en el listín telefónico. La eficiencia de un algoritmo de búsqueda estática depende de si el vector donde se busca está o no ordenado. En el caso del listín telefónico, buscar por nombre es fácil, pero buscar por número de teléfono es algo desesperante (al menos para los humanos). En esta sección, examinamos algunas soluciones al problema de búsqueda estática.

5.6.1 Búsqueda secuencial

Cuando el vector no está ordenado, tenemos pocas alternativas fuera de una *búsqueda secuencial* lineal, recorriendo todo el vector secuencialmente hasta que se encuentre el elemento buscado. La complejidad del algoritmo se analiza de tres formas. Primero, daremos el coste de una búsqueda sin éxito. Después veremos el coste en el caso peor de una búsqueda con éxito. Finalmente, veremos el coste en

el caso medio de una búsqueda con éxito. El analizar de forma separada los casos en los que la búsqueda tiene éxito es típico. También es típico el que las búsquedas sin éxito consuman más tiempo que las búsquedas con éxito (simplemente piense en la última vez que perdió algo en su casa). Para la búsqueda secuencial, el análisis es muy fácil.

La búsqueda secuencial es lineal.

Una búsqueda sin éxito requiere examinar cada uno de los elementos del vector, así que el tiempo será $O(N)$. En el peor caso, una búsqueda con éxito también requiere examinar cada elemento del vector porque podríamos no encontrar el elemento hasta la última posición. Luego el tiempo de ejecución en el caso peor en también lineal. En promedio, sin embargo, sólo buscamos en la mitad del vector. Es decir, para cada búsqueda con éxito en la posición i, hay una búsqueda con éxito correspondiente en la posición $N - i$ (suponiendo que empezamos a numerar desde 1). Sin embargo, $N/2$ sigue siendo $O(N)$. Como mencionamos anteriormente en este mismo capítulo, todos estos términos en notación O deberían ser, más correctamente, términos Θ. Sin embargo, el uso de la notación O es más popular.

5.6.2 Búsqueda binaria

Si el vector de entrada está ordenado, podemos utilizar la *búsqueda binaria*, la cual se realiza desde la mitad del vector, en vez de desde el final.

Si el vector de entrada está ordenado, entonces existe una alternativa a la búsqueda secuencial, la *búsqueda binaria*. La búsqueda binaria se realiza desde la mitad del vector, en vez de desde el final.

En todo momento, mantendremos dos variables `inicio` y `fin`, que delimitan la parte del vector en la que el elemento, si aparece, debe encontrarse. Inicialmente, el rango es desde 0 hasta $N - 1$. Si `inicio` es mayor que `fin`, sabemos que el elemento no está presente, así que lanzamos una excepción. En otro caso, igualamos la variable `medio` al punto medio en el rango actual (redondeando hacia abajo si el rango tiene un número par de elementos) y comparamos el elemento que estamos buscando con el elemento en la posición `medio`. Si hay éxito, hemos acabado. Si el elemento que estamos buscando es menor que el elemento en la posición `medio` entonces debería estar en el rango desde `inicio` hasta `medio -1`. Si es mayor, entonces debe encontrarse en el rango desde `medio +1` hasta `fin`. En la Figura 5.11, las líneas de la 18 a la 21 modifican el rango posible, esencialmente dividiéndolo por la mitad. Por el principio de divisiones sucesivas por la mitad, sabemos que el número de iteraciones será $O(\log N)$.

La *búsqueda binaria* es logarítmica porque el rango de búsqueda se divide por la mitad en cada iteración.

En una búsqueda sin éxito, el número de iteraciones del bucle es $\lfloor \log N \rfloor + 1$. Esto es debido a que el rango de búsqueda se divide por la mitad (redondeando hacia abajo si el rango tiene un número par de elementos) en cada iteración. Añadimos 1 porque el último rango abarca cero elementos. En una búsqueda con éxito, el caso peor consiste en $\lfloor \log N \rfloor$ iteraciones, porque en el caso peor reducimos la parte de vector hasta un único elemento. En el caso medio precisaremos sólo una iteración menos. Esto se debe a que la búsqueda de la mitad de los elementos nos conduce al caso peor, un cuarto de los elementos ahorran una búsqueda, y sólo uno entre 2^i elementos ahorrará i iteraciones con respecto al caso peor. Hemos pues de calcular una media ponderada, calculando la suma de una serie finita. El resultado es que el tiempo medio de ejecución para cada búsqueda es $O(\log N)$. En el Ejercicio 5.19 se pide completar este cálculo.

Para valores de N razonablemente grandes, la búsqueda binaria se ejecuta de forma más eficiente que la búsqueda secuencial. Por ejemplo, si N es 1.000, la

```
1  /**
2   * Realiza la búsqueda binaria estándar
3   * utilizando dos comparaciones por nivel.
4   * @exception ElementoNoEncontrado si es apropiado.
5   * @return índice donde se encuentra el elemento.
6   */
7  public static int busquedaBinaria( Comparable [ ] a,
8                                     Comparable x ) throws ElementoNoEncontrado
9  {
10     int inicio = 0;
11     int fin = a.length - 1;
12     int medio;
13
14     while( inicio <= fin )
15     {
16         medio = ( inicio + fin ) / 2;
17
18         if( a[ medio ].compara( x ) < 0 )
19             inicio = medio + 1;
20         else if( a[ medio ].compara( x ) > 0 )
21             fin = medio - 1;
22         else
23             return medio;
24     }
25
26     throw new ElementoNoEncontrado ( "La búsqueda binaria falla" );
27 }
```

Figura 5.11 Búsqueda binaria básica que distingue tres casos.

búsqueda secuencial requiere en promedio 500 comparaciones aproximadamente. En media, la búsqueda binaria requiere 8 iteraciones en una búsqueda con éxito (utilizando la fórmula anterior). Ya que cada iteración utiliza 1,5 comparaciones en promedio (unas veces 1 y otras 2), el total es de 12 comparaciones para una búsqueda con éxito. La búsqueda binaria mejora aún más en el caso peor o cuando las búsquedas no tienen éxito.

Optimizando la búsqueda binaria podríamos reducir el número de comparaciones a aproximadamente la mitad.

Si queremos hacer la búsqueda binaria aún más rápida, tenemos que hacer más ajustado el bucle más interno. Una estrategia posible es eliminar la comprobación (implícita) en el caso de una búsqueda con éxito del bucle más interno y reducir el rango a un solo elemento en todos los casos. Entonces podemos utilizar una comprobación más fuera del bucle para determinar si el elemento buscado está en el vector o no, como se muestra en la Figura 5.12. Si el elemento que estamos buscando en la Figura 5.12 no es mayor que el elemento en la posición `medio`, entonces está en el rango que incluye la posición `medio`. Cuando salimos del bucle, el rango es 1, y podemos comprobar si está ahí el elemento buscado.

En el algoritmo mejorado, el número de iteraciones es siempre $\lfloor \log N \rfloor$, pues siempre se reduce el rango a la mitad, posiblemente redondeando. El número de comparaciones necesarias es siempre $\lfloor \log N \rfloor + 1$.

La búsqueda binaria es sorprendentemente delicada de implementar. El Ejercicio 5.5 ilustra algunos errores comunes.

Observe que para valores N pequeños, como valores menores que 6, puede no merecer la pena la búsqueda binaria. Utiliza más o menos el mismo número de comparaciones para una búsqueda con éxito típica, pero tiene la sobrecarga de la línea 19 en cada iteración. En realidad, las últimas iteraciones de la búsqueda binaria pro-

```
1  /**
2   * Realiza la búsqueda binaria estándar
3   * utilizando una comparación por nivel.
4   * @exception ElementoNoEncontrado si es apropiado.
5   * @return índice donde se encuentra el elemento.
6   */
7  public static int busquedaBinaria( Comparable [ ] a, Comparable x )
8                                          throws ElementoNoEncontrado
9  {
10     if (a.length == 0) throw
11         new ElementoNoEncontrado ( "La búsqueda ha fallado" );
12
13     int inicio = 0;
14     int fin = a.length - 1;
15     int medio;
16
17     while( inicio < fin )
18     {
19         medio = ( inicio + fin ) / 2;
20
21         if( a[ medio ].compara( x ) < 0 )
22             inicio = medio + 1;
23         else
24             fin = medio;
25     }
26
27     if (a[ inicio ].compara( x ) == 0 )
28         return inicio;
29
30     throw new ElementoNoEncontrado ( "La búsqueda ha fallado" );
31 }
```

Figura 5.12 Búsqueda binaria básica que distingue dos casos.

gresan lentamente. Uno puede adoptar una estrategia híbrida en la cual el bucle de búsqueda binaria termina cuando el rango es pequeño, aplicándose entonces una búsqueda secuencial para terminar. De forma similar, la gente busca en un listín telefónico de forma no secuencial. Una vez que han reducido el rango de la búsqueda a una columna, realizan una búsqueda secuencial. La búsqueda en un listín telefónico no es secuencial, pero tampoco es binaria. Se discute en la próxima sección.

5.6.3 Búsqueda interpolada

La búsqueda binaria es muy rápida sobre un vector estático ordenado. De hecho, es tan rápida que difícilmente podríamos usar algo mejor. Sin embargo, un método de búsqueda estática que es en ocasiones más rápido es la *búsqueda interpolada*. Para que la búsqueda interpolada sea práctica, deben satisfacerse dos hipótesis:

1. Cada acceso debe ser muy costoso comparado con una instrucción típica. Por ejemplo, el vector podría estar en un disco en vez de en memoria principal, y cada comparación requiere un acceso a disco.
2. Los datos además de estar ordenados deben estar también uniformemente distribuidos. Por ejemplo, un listín telefónico está uniformemente distribuido. Por el contrario, si los elementos de la entrada son {1, 2, 4, 8, 16, ...}, entonces la distribución no sería uniforme.

Estas hipótesis son bastante restrictivas, de modo que es posible que en la práctica nunca nos convenga utilizar una búsqueda interpolada. Pero es interesante ver que hay más de una forma de resolver un problema, y que ningún algoritmo, incluso la clásica búsqueda binaria, es el mejor en todas las situaciones.

La idea básica de la búsqueda interpolada consiste en que estamos dispuestos a invertir más tiempo si con ello conseguimos una suposición más certera con respecto a donde podría encontrarse el elemento. La búsqueda binaria utiliza siempre el punto medio. Sin embargo, sería absurdo buscar a *Pablo Álvarez* en la mitad del listín telefónico; algún sitio cercano al principio sería claramente más apropiado. Por tanto, en vez de `medio`, utilizamos `proximo` para indicar el próximo elemento al que accederemos.

Veamos ahora un ejemplo de hasta qué punto la búsqueda interpolada podría ser buena. Supongamos que el rango de búsqueda contiene 1.000 elementos, el elemento menor en el rango es 1.000, el mayor es 1.000.000, y estamos buscando el valor 12.000. Si los elementos están uniformemente distribuidos, entonces podemos esperar encontrarlo en algún lugar cercano al duodécimo elemento. La fórmula aplicable es

$$próximo = inicio + \left\lceil \frac{x - a[inicio]}{a[fin] - a[inicio]} \times (fin - inicio - 1) \right\rceil.$$

El sustraer 1 es un ajuste técnico que se ha mostrado positivo en la práctica. Claramente, este cálculo es más costoso que el cálculo de la búsqueda binaria. Involucra una división extra (la división por 2 en la búsqueda binaria es simplemente un desplazamiento de un bit, igual que dividir por 10 es sencillo para las personas), una multiplicación y cuatro restas. Estos cálculos deben ser realizados utilizando operaciones de punto flotante. Una iteración podría incluso ser más lenta que la búsqueda binaria completa. Sin embargo, si el coste de estas operaciones es insignificante comparado con el coste de acceder a un elemento, entonces no es importante; sólo debemos preocuparnos del número de iteraciones.

La *búsqueda interpolada* tiene mejor cota O en promedio que la búsqueda binaria, pero tiene una utilidad limitada y un caso peor malo.

En el caso peor, cuando los datos no están uniformemente distribuidos, el tiempo de ejecución podría ser lineal y todos los elementos podrían tener que ser examinados. El Ejercicio 5.18 pide construir uno de estos casos. Sin embargo, bajo la suposición de que los elementos están razonablemente distribuidos, como en un listín telefónico, el número medio de comparaciones es $O(\log \log N)$. Esto significa que aplicamos el logaritmo dos veces seguidas. Para $N = 4$ billones, $\log N$ es aproximadamente 42, y $\log \log N$ es más o menos 5. Por supuesto, hay algunas constantes multiplicativas ocultas en la notación O, pero el logaritmo extra puede reducir considerablemente el número de iteraciones, mientras no se presente un caso malo. Sin embargo, probar el resultado de forma rigurosa es bastante complicado.

5.7 Comprobar el análisis de un algoritmo

Una vez que hemos efectuado el análisis de un algoritmo, queremos ver si es correcto y tan bueno como sea posible. Una forma de hacerlo es implementar el programa y ver si el tiempo de ejecución observado empíricamente se ajusta con el tiempo de ejecución predicho por el análisis.

Cuando N se incrementa por un factor de diez, el tiempo de ejecución se eleva en un factor de 10 en los programas lineales, de 100 en los programas cuadráticos y de 1.000 en los cúbicos. Los programas que se ejecutan en un tiempo $O(N \log N)$ tardarán algo más de diez veces más en las mismas circunstancias. Estos aumentos pueden ser difíciles de observar si los términos de orden inferior tienen coeficientes relativamente grandes y N no es suficientemente grande. Un ejemplo, es el salto de $N = 10$ a $N = 100$ en el tiempo de ejecución de las diferentes implementaciones para el problema del cálculo de la subsecuencia de suma máxima. También puede ser difícil diferenciar un programa lineal de uno $O(N \log N)$ basándose tan sólo en la evidencia empírica.

Otro truco utilizado a menudo para verificar si un programa es $O(F(N))$ es calcular los valores de $T(N)/F(N)$ para una serie de valores de N (generalmente espaciados por múltiplos de dos), donde $T(N)$ es el tiempo de ejecución observado empíricamente. Si $F(N)$ es una respuesta ajustada al tiempo de ejecución, entonces los valores calculados convergen a una constante positiva. Si $F(N)$ es una sobreestimación, los valores convergen a cero. Si $F(N)$ es una subestimación, y por tanto errónea, los valores divergen.

Por ejemplo, supongamos que escribimos un programa para realizar N búsquedas aleatorias utilizando el algoritmo de búsqueda binaria. Ya que cada búsqueda es logarítmica, esperamos que el tiempo de ejecución total del programa sea $O(N \log N)$. La Figura 5.13 muestra los valores observados para esta rutina, para varios tamaños de la entrada, sobre una computadora real. La tabla muestra que la última columna es la columna convergente, confirmando, por tanto, nuestra predicción, mientras que los números crecientes para T/N sugieren que $O(N)$ es una subestimación, y los valores rápidamente decrecientes de T/N^2 sugieren que $O(N^2)$ es una sobreestimación.

Observe en particular que no tenemos una convergencia absolutamente diáfana. Un problema es que el reloj que utilizamos para medir el tiempo del programa

N	**Tiempo de CPU T (milisegundos)**	T/N	T/N^2	$T/(N \log N)$
10.000	100	0,01000000	0,00000100	0,00075257
20.000	200	0,01000000	0,00000050	0,00069990
40.000	440	0,01100000	0,00000027	0,00071953
80.000	930	0,01162500	0,00000015	0,00071373
160.000	1.960	0,01225000	0,00000008	0,00070860
320.000	4.170	0,01303125	0,00000004	0,00071257
640.000	8.770	0,01370313	0,00000002	0,00071046

Figura 5.13 Tiempo de ejecución empírico para N búsquedas binarias en un vector de N elementos.

hace tictac sólo cada 10 ms. Observe también que no hay una gran diferencia entre $O(N)$ y $O(N \log N)$. Por supuesto, un algoritmo $O(N \log N)$ está más cerca de ser lineal que de ser cuadrático. Finalmente, tenga en cuenta que la máquina utilizada en el ejemplo tiene suficiente memoria para almacenar 640.000 objetos (en el caso de este experimento, enteros). Si esto no es cierto en su máquina, entonces no podrá reproducir similares resultados.

La próxima sección discute algunas de las limitaciones del análisis basado en la notación O.

5.8 Limitaciones del análisis *O*

El análisis O es una herramienta muy efectiva, pero es importante conocer sus limitaciones. Como ya se ha mencionado, no es apropiado para pequeñas cantidades de datos. Con pequeñas entradas, utilice el algoritmo más simple. También, para un algoritmo particular, la constante implícita en la notación O puede resultar demasiado grande en la práctica. Por ejemplo, si el tiempo de ejecución de un algoritmo está gobernado por la fórmula $2N \log N$ y otro tiene un tiempo de ejecución de $1.000N$, entonces el primer algoritmo será probablemente mejor, aunque su tasa de crecimiento sea mayor. Las constantes grandes pueden entrar en juego cuando un algoritmo es excesivamente complejo. También aparecen porque nuestro análisis no tiene en cuenta las constantes y por tanto no puede diferenciar cosas como accesos a memoria (que no son costosos) de accesos a disco (que típicamente son muchos miles de veces más costosos). Nuestro análisis supone memoria infinita, pero en las aplicaciones que involucran grandes conjuntos de datos, la falta de memoria suficiente puede ser un serio problema.

A veces, aun cuando las constantes y los términos no dominantes son tenidos en cuenta, se prueba empíricamente que el análisis era una sobreestimación. En tal caso, el análisis debe ser ajustado (generalmente mediante una observación más inteligente). Otras veces la cota de tiempo de ejecución en promedio puede ser significativamente menor que la cota en el caso peor, y por tanto ninguna mejora en la cota es posible. Hay muchos algoritmos complicados para los cuales la cota en el caso peor se alcanza para una mala entrada, pero en la práctica es generalmente una sobreestimación. Dos ejemplos son los métodos de ordenación Shellsort y quicksort (ambos se discuten en el Capítulo 8).

El caso peor es en ocasiones poco representativo por lo que puede ser ignorado sin problemas. Otras veces, es muy común y no puede ser ignorado.

Sin embargo, las cotas en el caso peor son normalmente más fáciles de obtener que sus contrapartidas en el caso medio. Por ejemplo, hasta la fecha no se ha podido obtener un análisis matemático del tiempo de ejecución en el caso medio de Shellsort. A veces, incluso definir con exactitud lo que significa «caso medio» es difícil. Utilizamos un análisis en el caso peor porque es conveniente, y también porque, en la mayoría de los casos, el análisis en el caso peor es bastante significativo. Al realizar el análisis, frecuentemente podremos decir si también sería aplicable al caso promedio.

El análisis en el caso medio es casi siempre mucho más difícil que en el caso peor.

Resumen

En este capítulo se ha introducido el amplio tema del análisis de algoritmos y se ha mostrado que decisiones algorítmicas generalmente influyen más en el tiempo

de ejecución de un programa que los trucos de programación. También se ha mostrado la enorme diferencia entre los algoritmos lineales y cuadráticos y se ha visto cómo los algoritmos cúbicos son, en la mayor parte, insatisfactorios. Se ha examinado un algoritmo que podría tomarse como base para nuestra primera estructura de datos. La búsqueda binaria soporta de forma eficiente operaciones estáticas (es decir, búsqueda pero no actualización), proporcionando por tanto una búsqueda logarítmica en el caso peor. Capítulos posteriores del texto examinan estructuras de datos dinámicas que soportan eficientemente las actualizaciones (tanto inserción como eliminación).

En el próximo capítulo definimos las estructuras de datos y sus operaciones permitidas. También se detiene en algunas aplicaciones de las estructuras de datos y discute su eficiencia.

Elementos del juego

algoritmo lineal Un algoritmo cuyo tiempo de ejecución crece como $O(N)$. Si el tamaño de la entrada aumenta por un factor f, entonces el tiempo de ejecución también aumenta por el mismo factor. Es el tiempo de ejecución más favorable para un algoritmo.

búsqueda binaria El método de búsqueda utilizado cuando el vector de entrada está ordenado. Las búsquedas se realizan desde la mitad en vez desde el final. La búsqueda binaria es logarítmica porque el rango de búsqueda se reduce a la mitad en cada iteración.

búsqueda estática Encuentra un elemento en un conjunto de datos que nunca es modificado.

búsqueda interpolada Algoritmo de búsqueda estática que tiene mejor rendimiento O en el caso medio que la búsqueda binaria, pero que sólo se puede utilizar en casos limitados y tiene una complejidad mala en el caso peor.

búsqueda secuencial Método lineal de búsqueda que recorre el vector hasta que encuentra el elemento.

cota en el caso medio La cota en la cual el tiempo de ejecución se obtiene mediante la media sobre todas las posibles entradas de tamaño N.

cota en el caso peor Una garantía del coste sobre todas las entradas de cierto tamaño.

logaritmo El logaritmo de N (en base 2) es el valor X tal que 2 elevado a X es igual a N.

notación Ω Notación similar al mayor o igual cuando consideramos tasas de crecimiento.

notación Θ Notación similar al igual cuando consideramos tasas de crecimiento.

notación o Notación similar al menor estricto cuando se consideran tasas de crecimiento.

notación O Notación utilizada para capturar el término más dominante de una función. La notación O es similar al menor o igual cuando consideramos tasas de crecimiento.

números armónicos El número armónico N-ésimo es la suma de los inversos de los N primeros enteros positivos. La tasa de crecimiento de los números armónicos es logarítmica.

principio de duplicaciones repetidas Principio por el cual si empezamos con 1, podemos multiplicar por 2 sólo un número logarítmico de veces, hasta alcanzar N.

principio de sucesivas divisiones por la mitad Principio por el cual si empezamos con N, podemos dividir de forma repetida por la mitad solamente un número de veces logarítmico, hasta alcanzar 1. Se utiliza para obtener rutinas de búsqueda logarítmicas.

subcuadrático Algoritmo cuyo tiempo de ejecución es estrictamente menor que cuadrático. El tiempo de ejecución puede ser escrito como $o(N^2)$.

Errores comunes

1. En el caso de bucles anidados, el tiempo total es función del producto del tamaño de los bucles. En el caso de bucles consecutivos, no.
2. No cuente simplemente el número de bucles. Dos bucles anidados, cada uno de los cuales se ejecuta desde 1 hasta N^2, representan un tiempo $O(N^4)$.
3. No escriba expresiones como $O(2N^2)$ o $O(N^2 + N)$. Sólo es necesario el término dominante, sin la constante multiplicativa.
4. Utilice igualdades con la notación O, Ω, etc. No escriba que el tiempo de ejecución es $>O(N^2)$; esto no tiene sentido porque la notación O es una cota superior. No escriba que el tiempo de ejecución es $<O(N^2)$; si el propósito es decir que el tiempo de ejecución es estrictamente menor que cuadrático, utilice la notación o.
5. Utilice la notación Ω, y no la notación O, para una cota inferior.
6. Utilice el logaritmo para describir el tiempo de ejecución de un problema resuelto dividiendo por la mitad su tamaño en tiempo constante. No obstante si tarda más de un tiempo constante en dividir por la mitad el tamaño, el logaritmo no se aplica.
7. La base del logaritmo no es importante cuando se utiliza la notación O. Es por tanto innecesario precisarla, aunque por ello mismo podemos elegir en cada caso la base más favorable para nuestros razonamientos.

En Internet

Los tres algoritmos para la obtención de la subsecuencia de suma máxima, así como un cuarto tomado de la Sección 7.5, están disponibles, junto con una función `main` que realiza los tests de tiempo de ejecución. También se proporciona un algoritmo de búsqueda binaria en el paquete `DataStructures`. He aquí los nombres de ficheros y los directorios:

BinarySearch.java	Traducido por `BusquedaBinaria.java`, contiene el algoritmo de la búsqueda binaria de la Figura 5.12, aunque ligeramente modificado. Se encuentra en el directorio **DataStructures**. La Figura 5.11 se encuentra en el directorio **Chapter05**.
MaxSumTest.java	Traducido por `TestSumaMax.java`, contiene cuatro algoritmos para el problema de la subsecuencia de suma máxima. Se encuentra en el directorio **Chapter05**.

Ejercicios

Cuestiones breves

5.1. Se extraen tres bolas de una caja como se especifica en el Teorema 5.1 obteniéndose las combinaciones desde *a*) hasta *d*). ¿Cuáles son los correspondientes valores de i, j y k?

a) Roja, 5, 6.
b) Azul, 5, 6.
c) Azul, 3, Roja.
d) 6, 5, Roja.

5.2. ¿Por qué no es suficiente una implementación basada exclusivamente en el Teorema 5.2 para obtener un tiempo de ejecución subcuadrático para el problema de la obtención de la subsecuencia de suma máxima?

5.3. Suponga que $T_1(N) = O(F(N))$y $T_2(N) = O(F(N))$. ¿Cuáles entonces de las siguientes afirmaciones son ciertas?

a) $T_1(N) + T_2(N) = O(F(N))$.
b) $T_1(N) - T_2(N) = O(F(N))$.
c) $T_1(N)/T_2(N) = O(1)$.
d) $T_1(N) = O(T_2(N))$.

5.4. Se analizan dos programas A y B y se ve que en el caso peor tienen tiempos de ejecución no mayores que $150N \log N$ y N^2, respectivamente. Responda a las siguientes cuestiones, cuando ello sea posible:

a) ¿Qué programa da la mejor garantía en tiempo de ejecución para valores grandes de N ($N > 10.000$)?
b) ¿Qué programa da la mejor garantía en tiempo de ejecución para valores pequeños de N ($N < 100$)?
c) ¿Qué programa se ejecutará más rápido *en promedio* para $N = 1.000$?
d) ¿Es posible que el programa B se ejecute más rápidamente que el programa A en *todas* las posibles entradas?

5.5. Para la rutina de búsqueda binaria de la Figura 5.11, muestre las consecuencias de las siguientes modificaciones en el código:

a) Línea 14: utilizando el test `inicio < fin`.
b) Línea 16: haciendo la asignación `medio = inicio + fin / 2`.
c) Línea 19: asignando `inicio = medio`.
d) Línea 21: asignando `fin = medio`.

Problemas teóricos.

5.6. Determine, para los algoritmos típicos que utiliza para realizar cálculos a mano, el tiempo de ejecución en hacer los siguiente:

a) Sumar dos enteros de N dígitos.
b) Multiplicar dos enteros de N dígitos.
c) Dividir dos enteros de N dígitos.

5.7. En términos de N, ¿cuál es el tiempo de ejecución del siguiente algoritmo para calcular X^N?

```
public static double potencia( double x, int n )
{
    double resultado = 1.0;

    for( int i = 0; i < n; i++ )
        resultado *= x;
    return resultado;
}
```

5.8. Evalúe directamente el triple sumatorio que precede al Teorema 5.1. Verifique que la respuesta es idéntica a la allí obtenida.

5.9. Para el algoritmo cuadrático, determine con precisión cuántas veces se ejecuta el bucle más interno.

5.10. Un algoritmo tarda 0,5 milisegundos para una entrada de tamaño 100. ¿Cuánto tardará con una entrada de tamaño 500 si el tiempo de ejecución es el siguiente?

a) Lineal.
b) $O(N \log N)$.
c) Cuadrático.
d) Cúbico.

5.11. Un algoritmo tarda 0,5 milisegundos para una entrada de tamaño 100. ¿Qué tamaño de entrada puede procesar en un minuto, si el tiempo de ejecución es el siguiente?

a) Lineal.
b) $O(N \log N)$.
c) Cuadrático.
d) Cúbico.

5.12. Complete la tabla de la Figura 5.10 con estimaciones para tiempos de ejecución que son demasiado grandes para ser simulados. Interpole los tiempos de ejecución para los cuatro algoritmos y estime el tiempo necesario para calcular la subsecuencia de suma máxima de un millón de números. ¿Qué hipótesis ha asumido?

5.13. Ordene las siguientes funciones por tasa de crecimiento: N, $\sqrt{N}$, $N^{1,5}$, N^2, $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log (N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, N^3 y $N^2 \log N$. Indique qué funciones crecen a la misma velocidad.

5.14. Para cada uno de los siguientes fragmentos de programas, haga lo siguiente:

a) Dé un análisis en notación O del tiempo de ejecución.
b) Implemente el código y ejecútelo para diferentes valores de N.
c) Compare su análisis con los tiempos de ejecución reales.

```
// Fragmento #1
for( int i = 0; i < n; i++ )
    suma++;
```

```
// Fragmento #2
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++)
        suma++;

// Fragmento #3
for( int i = 0; i < n; i++ )
    suma++;
for( int j = 0; j < n; j++)
    suma++;

// Fragmento #4
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++)
        suma++;

// Fragmento #5
for( int i = 0; i < n; i++ )
    for( int j = 0; j < i; j++)
        suma++;

// Fragmento #6
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++)
        for( int k = 0; k < j; k++)
            suma++;
```

5.15. En ocasiones, multiplicar el tamaño de los bucles anidados puede hacer que el tiempo de ejecución en notación O sea una sobreestimacion. Repita el Ejercicio 5.14 para el siguiente fragmento de programa:

```
for( int i = 1; i < n; i++ )
    for( int j = 0; j < i * i; j++ )
        if( j % i == 0 )
            for( int k = 0; k < j; k++ )
                suma++;
```

5.16. En un juicio reciente, un juez condenó a una ciudad por desacato y ordenó una multa de 2 euros para el primer día. Cada día posterior, hasta que la ciudad acatara la orden del juez, la multa sería elevada al cuadrado (es decir, la multa aumentaría de la siguiente manera: 2 euros, 4 euros, 16 euros, 256 euros, 65536 euros, ...).

a) ¿Cuál sería la multa el día N?

b) ¿Cuántos días harían falta para que la multa alcanza E euros? (Una respuesta en notación O será suficiente.)

5.17. Demuestre el Teorema 5.5. *Pista*: Muestre que $\sum_2^N \frac{1}{i} < \int_1^N \frac{dx}{x}$. Obtenga luego de forma similar una cota inferior.

5.18. Construya un ejemplo en el cual una búsqueda interpolada examine cada elemento del vector.

5.19. Analice el coste en promedio de una búsqueda con éxito para el algoritmo de búsqueda binaria.

Problemas prácticos

5.20. Escriba un algoritmo eficiente para determinar si existe un entero i tal que $A_i = i$ en un vector de enteros ordenados de forma creciente. ¿Cuál es su tiempo de ejecución?

5.21. Un número primo no tiene divisores aparte de 1 y de él mismo. Se pide lo siguiente:

a) Escriba un programa para determinar si un entero positivo N es primo. En términos de N, ¿cuál es el tiempo de ejecución en el caso peor de su programa?

b) Sea B el número de bits en la representación binaria de N. ¿Cuál es el valor de B?

c) En términos de B, ¿cuál es el tiempo de ejecución en el caso peor de su programa?

d) Compare los tiempos de ejecución necesarios para determinar si un número de 20 bits y otro de 40 bits son primos.

5.22. Un problema importante en análisis numérico consiste en encontrar una solución a la ecuación $F(X) = 0$ para un F arbitrario. Si la función es continua y tenemos dos puntos *bajo* y *alto* para los cuales $F(bajo)$ y $F(alto)$ tienen signo opuesto, entonces debe existir una raíz entre *bajo* y *alto*, que puede ser encontrada bien mediante una búsqueda binaria o con una búsqueda interpolada. Escriba una función que tome como parámetros F, *bajo* y *alto* y encuentre una raíz de la ecuación. ¿Qué debe hacer para asegurar la terminación?

5.23. Un elemento mayoritario en un vector A de tamaño N es un elemento que aparece más de $N/2$ veces (en consecuencia habrá a lo sumo uno). Por ejemplo, el vector

3, 3, 4, 2, 4, 4, 2, 4, 4

contiene un elemento mayoritario (4), mientras que el vector

3, 3, 4, 2, 4, 4, 2, 4

no tiene elemento mayoritario. Escriba un algoritmo para encontrar el elemento mayoritario cuando exista, o que diga que no exista. ¿Cuál es el tiempo de ejecución de su algoritmo? (Hay una solución $O(N)$.)

Prácticas de programación

5.24. La criba de Eratóstenes es un método utilizado para calcular todos los primos menores que N. Empiece haciendo una tabla de enteros desde 2 hasta N. Encuentre el menor entero i que no esté tachado. Entonces imprima i y tache i, $2i$, $3i$, etc. Cuando $i > \sqrt{N}$, el algoritmo termina. Se ha probado que el tiempo de ejecución es $O(N \log \log N)$. Escriba un programa para implementar la criba y verifique que el tiempo de ejecución es en efecto el señalado. ¿Qué dificultad ha encontrado para distinguir el tiempo de ejecución entre $O(N)$ y $O(N \log N)$?

5.25. La ecuación $A^5 + B^5 + C^5 + D^5 + E^5 = F^5$ tiene exactamente una solución entera que satisfaga $0 < A \leqslant B \leqslant C \leqslant D \leqslant E \leqslant F \leqslant 75$. Escriba un programa para encontrar dicha solución. *Pista*: primero, calcule todos los valores de X^5 y almacénelos en un vector. Entonces, para cada tupla (A, B, C, D, E), sólo necesita comprobar si existe algún F en el vector. Hay varias formas de buscar F. Un método consiste en aplicar búsqueda binaria. Sin embargo podríamos probar que otros métodos pueden ser más eficientes.

5.26. Implemente los algoritmos para el problema de la obtención de la subsecuencia de suma máxima, para obtener datos equivalentes a los de la Figura 5.10. Compile los programas con las opciones que conduzcan a una mayor eficiencia.

Bibliografía

El problema de la obtención de la subsecuencia de suma máxima se ha tomado de [5]. Los libros [4], [5] y [6] muestran cómo optimizar la velocidad de los programas. La búsqueda interpolada fue sugerida por primera vez en [14] y analizada en [13]. Los libros [1], [8] y [17] proporcionan un tratamiento más riguroso del análisis de algoritmos. La serie de tres partes [10], [11] y [12], actualizada recientemente, sigue siendo la principal referencia en el tema.

Los conocimientos matemáticos necesarios para el análisis de algoritmos más avanzado se proporciona en [2], [3], [7], [15] y [16]. Un libro especialmente bueno de análisis avanzado es [9].

1. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).
2. M. O. Albertson y J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, NY (1988).
3. Z. Bavel, *Math Companion for Computer Science*, Reston Publishing Company, Reston, Va (1982).
4. J. L. Bentley, *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, NJ (1982).
5. J. L. Bentley, *Programming Pearls*, Addison-Wesley. Reading, Mass. (1986).
6. J. L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, Mass. (1988).
7. R. A. Brualdi, *Introductory Combinatorics*, North-Holland, New York, NY (1977).
8. T. H. Cormen, C. E. Leiserson, y R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass. (1990).
9. R. L. Graham, D. E. Knuth, y O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, Mass. (1989).
10. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, 3.ª ed., Addison-Wesley, Reading, Mass. (1997).

11. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms,* 3.ª ed., Addison-Wesley, Reading. Mass. (1997).

12. D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching,* 2.ª ed., Addison-Wesley, Reading, Mass. (1997).

13. Y. Pearl, A. Itai, y H. Avni, «Interpolation Search—A log log N Search», *Communications of the ACM* **21** (1978), 550-554.

14. W. W. Peterson, «Addressing for Random Storage», *IBM Journal of Research and Development* **1** (1957), 131-132.

15. F. S. Roberts, *Applied Combinatorics*, Prentice-Hall, Englewood Cliffs, NJ (1984).

16. A. Tucker, *Applied Combinatorics*, 2.ª ed., John Wiley & Sons, New York, NY (1984).

17. M. A. Weiss, *Data Structures and Algorithm Analysis in C*, 2.ª ed., Benjamin/Cummings, Redwood City, Calif. (1997).