Capítulo 5

Análisis de algoritmos

En la primera parte hemos examinado cómo nos puede ayudar la programación orientada a objetos durante el diseño y la implementación de sistemas de gran tamaño. No nos hemos fijado en las cuestiones de rendimiento. Generalmente, si utilizamos una computadora es porque necesitamos procesar una gran cantidad de datos. Y cuando ejecutamos un programa con una gran cantidad de datos, tenemos que estar seguros de que el programa termine su tarea en un tiempo razonable. Aunque el tiempo total de ejecución depende en alguna medida del lenguaje de programación que empleemos y en menor medida de la metodología utilizada (como por ejemplo programación procedimental en lugar de orientada a objetos), a menudo esos factores son constantes del diseño que no se pueden modificar. Aún así, de lo que más depende el tiempo de ejecución es de la elección de los algoritmos.

Un *algoritmo* es un conjunto claramente especificado de instrucciones que la computadora seguirá para resolver un problema. Una vez que se proporciona un algoritmo para un problema y se verifica que es correcto, el siguiente paso consiste en determinar la cantidad de recursos, como por ejemplo tiempo y espacio de memoria, que el algoritmo requerirá. Este paso se denomina *análisis de algoritmos*. Un algoritmo que requiera varios cientos de gigabytes de memoria principal no será útil para la mayoría de las máquinas actuales, incluso aunque sea completamente correcto.

En este capítulo, vamos a abordar las siguientes cuestiones:

- Cómo estimar el tiempo requerido por un algoritmo.
- Cómo utilizar técnicas que reduzcan drásticamente el tiempo de ejecución de un algoritmo.
- Cómo emplear un marco de análisis matemático que describa con el mayor rigor posible el tiempo de ejecución de un algoritmo.
- Cómo describir una rutina simple de *búsqueda binaria*.

5.1 ¿Qué es el análisis de algoritmos?

La cantidad de tiempo que cualquier algoritmo tarda en ejecutarse depende casi siempre de la cantidad de entrada que deba procesar. Cabe esperar, por ejemplo, que ordenar 10.000 elementos requiera más tiempo que ordenar 10 elementos. El tiempo de ejecución de un algoritmo es, por tanto, una función del tamaño de la entrada. El valor exacto de la función dependerá de muchos

Una mayor cantidad de datos implica que el programa necesita más tiempo de ejecución.

factores, como por ejemplo de la velocidad de la máquina utilizada, de la calidad del compilador y, en algunos casos, de la calidad del programa. Para un determinado programa en una determinada computadora, podemos dibujar en una gráfica la función que expresa el tiempo de ejecución. La Figura 5.1 ilustra una de esas gráficas para cuatro programas distintos. Las curvas representan cuatro funciones comúnmente encontradas 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 varían entre 0 y 10 microsegundos. Un rápido vistazo a la Figura 5.1, y a su compañera, la Figura 5.2, sugiere que las curvas lineal, $O(N \log N)$, cuadrática y cúbica representan distintos tiempos de ejecución, en orden de preferencia decreciente.

De las funciones que comúnmente nos podemos encontrar en el análisis de algoritmos, las *lineales* representan los algoritmos más eficientes.

Un ejemplo sería el problema de descargar un archivo de Internet. Suponga que hay un retardo inicial de 2 segundos (para establecer una conexión), después de lo cual la descarga se produce una velocidad de 160 K/seg. Entonces, si el archivo tiene N kilobytes, el tiempo para la descarga está dado por la fórmula $T(N) = N/160 + 2$. Esta es una *función lineal*. Descargar un archivo de 8.000K requiere aproximadamente 52 segundos, mientras que descargar un archivo dos veces mayor (16.000K) requiere unos 102 segundos, lo que aproximadamente es el doble de tiempo. Esta propiedad en la que el tiempo es, en esencia, directamente proporcional al tamaño de la entrada, es la característica fundamental de un *algoritmo lineal*, que es el tipo de algoritmo más eficiente. Por contraste, como muestran estas dos primeras gráficas, algunos de los algoritmos no lineales requieren tiempos de ejecución muy grandes. Por ejemplo, el algoritmo lineal es mucho más eficiente que el algoritmo cúbico.

En este capítulo, vamos a considerar varias cuestiones importantes:

- ¿Es importante disponer siempre de la curva más eficiente?
- ¿Cómo podemos comparar unas curvas con otras?

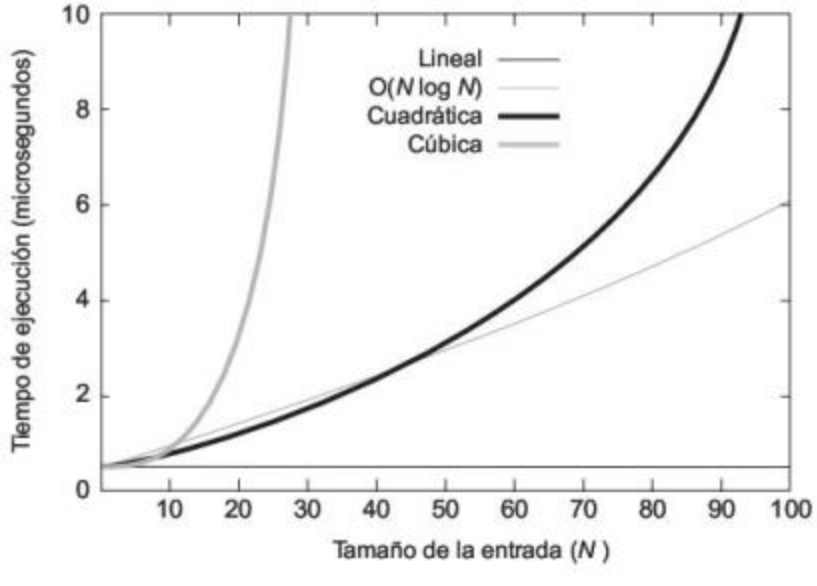

Figura 5.1 Tiempos de ejecución para entradas de pequeño tamaño.

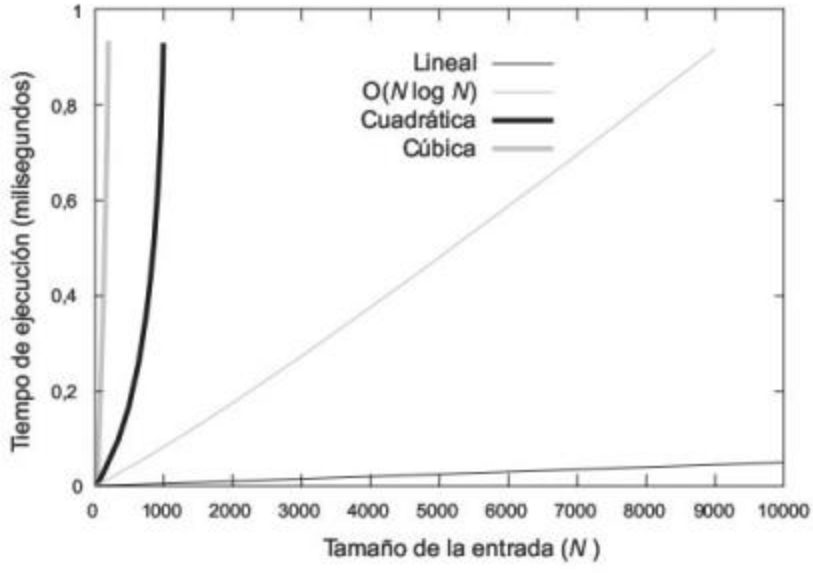

Figura 5.2 Tiempos de ejecución para entradas de tamaño moderado.

- ¿Cómo podemos determinar a qué curva corresponde un determinado algoritmo?
- ¿Cómo podemos diseñar algoritmos que eviten mostrar alguna de las curvas menos eficientes?

Una *función cúbica* es una función cuyo término dominante es una cierta constante multiplicada por N^3. Por ejemplo, $10N^3 + N^2 + 40N + 80$ sería una función cúbica. De forma similar, una función cuadrática tiene un término dominante que es igual a una constante multiplicada por N^2, mientras que una función lineal tiene un término dominante que es igual a una constante multiplicada por N. La expresión $O(N \log N)$ 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 la típica base 2) es solo 20. El logaritmo crece más lentamente que una raíz cuadrada o cúbica (o cualquier otra raíz). Hablaremos del logaritmo con más profundidad en la Sección 5.5.

Si tenemos dos funciones, cualquiera de ellas puede ser más pequeña que la otra en un punto determinado, así que afirmar que, por ejemplo $F(N) < G(N)$ no tiene sentido. En lugar de ello, lo que hacemos es medir la tasa de crecimiento de las funciones. Esto está justificado por tres razones distintas. En primer lugar, para las funciones cúbicas, como la que se muestra en la Figura 5.2, cuando N es 1.000 el valor de la función cúbica está casi completamente determinado por el término cúbico. En la función $10N^3 + N^2 + 40N + 80$, para $N = 1.000$, el valor de la función es 10.001.040.080, del cual 10.000.000.000 se debe al término $10N^3$. Si utilizáramos únicamente el término cúbico para estimar el valor de la función completa, el error que se produciría sería de aproximadamente del 0,01 por ciento. Para un valor de N suficientemente grande, el valor de una función está principalmente determinado por su término dominante (el significado de *suficientemente grande* varía según la función).

La tasa de crecimiento de una función tiene su máxima importancia cuando N es suficientemente grande.

La segunda razón por la que medimos la tasa de crecimiento de las funciones es que el valor exacto de la constante que multiplica al término dominante no tienen ningún significado si tratamos

de comparar unas máquinas con otras (aunque los valores relativos de esa constante para funciones que crezcan de manera idéntica sí que pueden ser significativos). Por ejemplo, la calidad del compilador podría tener una gran influencia sobre el valor de esa constante. La tercera razón es que los valores pequeños de N no suelen ser importantes. Para $N = 20$, la Figura 5.1 muestra que todos los algoritmos terminan en los 5 μs. La diferencia entre el algoritmo mejor y el peor es inferior al tiempo que tardamos en parpadear.

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

Utilizamos la notación *O mayúscula* o notación de Landau para capturar el término más dominante de una función y para representar la tasa de crecimiento. Por ejemplo, el tiempo de ejecución de un algoritmo cuadrático se especifica como $O(N^2)$ (que se lee "de orden ene cuadrado"). La notación O mayúscula también nos permite establecer un orden relativo entre funciones, comparando los términos dominantes. Veremos la notación O mayúscula de manera más formal en la Sección 5.4.

Para valores pequeños de N (por ejemplo, inferiores a 40), la Figura 5.1 muestra que una curva puede ser inicialmente mejor que otra y que sin embargo eso no se cumple para valores de N más grandes. Por ejemplo, la curva cuadrática es inicialmente mejor que la curva $O(N \log N)$, pero a medida que N se hace suficientemente grande, el algoritmo cuadrático pierde su ventaja. Para pequeñas cantidades de entrada, hacer comparaciones entre funciones resulta difícil, porque las constantes multiplicativas tienen un valor muy significativo. La función $N + 2.500$ es mayor que N^2 cuando N es menor que 50. Pero a medida que crece N, la función lineal siempre termina por tener un valor inferior que la función cuadrática. Además, lo que es aún más importante, para pequeños tamaños de entrada los tiempos de ejecución suelen ser irrelevantes, así que no necesitamos preocuparnos por ellos. Por ejemplo, la Figura 5.1 muestra que cuando N es inferior a 25, los cuatro algoritmos se ejecutan en menos de 10 μs. En consecuencia, cuando los tamaños de entrada son muy pequeños, una buena regla práctica consiste en utilizar el algoritmo que sea más simple.

La Figura 5.2 demuestra claramente las diferencias entre las distintas curvas para tamaños de entrada grandes. Un algoritmo lineal resuelve el problema de tamaño 10.000 en una pequeña fracción de segundo. El algoritmo $O(N \log N)$ utiliza un tiempo aproximadamente 10 veces mayor. Observe que las diferencias de tiempo reales dependerán de las constantes implicadas, y pueden por tanto ser mayores o menores de las indicadas. Dependiendo de estas constantes, un algoritmo $O(N \log N)$ podría ser más rápido que un algoritmo lineal para tamaños de entrada relativamente grandes. Sin embargo, para algoritmos de igual complejidad, los algoritmos lineales tienden a tener un mejor rendimiento que los algoritmos $O(N \log N)$.

Los algoritmos cuadráticos no resultan prácticos para tamaños de entrada superiores a unos pocos miles.

Sin embargo, esta relación no es cierta para los algoritmos cuadráticos y cúbicos. Los algoritmos cuadráticos no suelen ser nunca prácticos cuando el tamaño de entrada supera unos pocos miles, mientras que los algoritmos cúbicos dejan de ser prácticos para tamaños de entrada de solo unos pocos centenares. Por ejemplo, no es práctico emplear un algoritmo de ordenación sencillo para un millón elementos, porque los algoritmos de ordenación más simples (como la ordenación por selección o el algoritmo de la burbuja) son algoritmos cuadráticos. Los algoritmos de ordenación presentados en el Capítulo 8 se ejecutan en un tiempo *subcuadrático*; es decir, mejor que $O(N^2)$, lo cual hace que sea práctico ordenar matrices de gran tamaño.

La característica más llamativa de estas curvas es que los algoritmos cuadráticos y cúbicos no son competitivos en comparación con los otros para entradas razonablemente grandes. Podemos

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 ordenadas de menor a mayor tasa de crecimiento.

codificar el algoritmo cuadrático utilizando un lenguaje máquina altamente eficiente y no molestarnos apenas en la codificación del algoritmo lineal, y a pesar de todo el algoritmo cuadrático saldrá perdiendo. Ni siquiera los más inteligentes trucos de programación pueden hacer que se ejecute rápidamente un algoritmo no eficiente. Por tanto, antes de perder el tiempo tratando de optimizar el código, lo primero que hay que hacer es optimizar el algoritmo. La Figura 5.3 muestra las funciones que describen comúnmente los tiempos de ejecución de los algoritmos por orden de menor a mayor tasa de crecimiento.

Los algoritmos cúbicos no resultan prácticos para tamaños de entrada de solo unos pocos centenares.

5.2 Ejemplos de tiempos de ejecución de diversos algoritmos

En esta sección vamos a examinar tres problemas. También esbozaremos algunas posibles soluciones y determinaremos el tiempo de ejecución que exhibirán los algoritmos, sin proporcionar programas detallados. El objetivo de esta sección es proporcionar al lector una cierta intuición acerca del análisis de algoritmos. En la Sección 5.3 daremos más detalles sobre el proceso y en la Sección 5.4 abordaremos formalmente un problema de análisis de algoritmos.

Vamos a examinar los siguientes problemas en esta sección:

Elemento mínimo de una matriz
Dada una matriz de N elementos, determinar el menor de ellos.

Puntos más próximos en el plano
Dados N puntos en un plano (es decir, en un sistema de coordenadas x-y), encontrar la pareja de puntos más próximos.

Puntos colineales en el plano
Dados N puntos en un plano (es decir, en un sistema de coordenadas x-y), determinar si cualesquiera tres puntos forman una línea recta.

El problema del elemento mínimo es fundamental en informática. Se puede resolver de la forma siguiente:

1. Mantener una variable `min` que almacene el elemento mínimo.
2. Inicializar `min` con el valor del primer elemento.
3. Hacer un barrido secuencial de la matriz y actualizar `min` de la forma apropiada.

El tiempo de ejecución de este algoritmo será $O(N)$ o lineal, porque lo que hacemos es repetir una cantidad fija de trabajo para cada elemento de la matriz. Un algoritmo lineal es lo mejor que podemos esperar. Si en este caso se consigue es porque tenemos que examinar cada elemento de la matriz, lo cual es un proceso que requiere un tiempo lineal.

El problema de los puntos más próximos es un problema fundamental en gráficos, que se puede resolver de la forma siguiente:

1. Calcular la distancia entre cada pareja de puntos.
2. Quedarse con la distancia mínima.

Sin embargo, este cálculo es muy costoso, porque hay $N(N-1)/2$ pares de puntos.[1] Por tanto, hay aproximadamente N^2 pares de puntos. Examinar todos estos pares y hallar la distancia mínima entre ellos requiere un tiempo cuadrático. Existe un algoritmo mejor que se ejecuta en un tiempo $O(N \log N)$ y funciona por el procedimiento de evitar tener que calcular todas las distancias. También existe un algoritmo que requiere un tiempo $O(N)$. Estos dos algoritmos utilizan una serie de sutiles observaciones para proporcionar resultados más rápidamente, y caen fuera del alcance de este texto.

El problema de los puntos colineales es importante para muchos algoritmos gráficos. La razón es que la existencia de puntos colineales introduce un caso degenerado que requiere un tratamiento especial. El problema se puede resolver directamente enumerando todos los grupos de tres puntos. Esta solución es aún más cara, desde el punto de vista computacional, que la del problema de los puntos más próximos, porque el número de grupos de tres puntos distintos es $N(N-1)(N-2)/6$ (utilizando un razonamiento similar al empleado en el problema de los puntos más próximos). Este resultado nos dice que la solución directa se conseguiría en un algoritmo cúbico. También existe una estrategia más inteligente (que también queda fuera del alcance de este texto) que permite resolver el problema en un tiempo cuadrático (y actualmente se está investigando activamente de manera continua para conseguir mejoras adicionales).

En la Sección 5.3 examinaremos un problema que ilustra las diferencias entre los algoritmos lineales, cuadráticos y cúbicos. También mostraremos cómo se compara el rendimiento de estos algoritmos con una predicción matemática. Finalmente, después de explicar las ideas básicas, examinaremos de manera más formal la notación *O mayúscula*.

[1] Cada uno de los N puntos puede emparejarse con $N-1$ puntos, lo que nos da un total de $N(N-1)$ parejas. Sin embargo, este emparejamiento hace que se cuenten dos veces las parejas A, B y B, A, así que es preciso dividir entre 2.

5.3 El problema de la suma máxima de una subsecuencia contigua

En esta sección vamos a considerar el siguiente problema:

> **Problema de la suma máxima de una subsecuencia contigua**
> Dada una serie de enteros (posiblemente negativos) $A_1, A_2, \ldots, A_N$, encontrar el valor máximo de $\sum_{k=i}^{j} A_k$ e identificar la secuencia correspondiente. La suma máxima de una secuencia contigua es cero si todos los enteros son negativos.

Por ejemplo, si la entrada es {–2, **11**, **–4**, **13**, –5, 2}, entonces la respuesta es 20, que representa la subsecuencia contigua que abarca los elementos 2 a 4 (mostrados en negrita). Como segundo ejemplo, para la entrada{ 1, –3, **4**, **–2**, **–1**, **6**}, la respuesta es 7 para la subsecuencia que abarca los últimos cuatro elementos.

Los detalles de programación se toman en consideración después del diseño del algoritmo.

En Java, las matrices comienzan con cero, por lo que un programa Java representaría la entrada como una secuencia A_0 a A_{N-1}. Esto no es más que un detalle de programación y no forma parte del diseño de un algoritmo.

Considere siempre las soluciones vacías.

Antes de analizar los algoritmos existentes para resolver este problema, necesitamos comentar algo acerca del caso degenerado en el que todos los enteros de entrada sean negativos. El enunciado del problema nos da una suma máxima de subsecuencia contigua igual a 0 para este caso. Podríamos preguntarnos por qué se hace esto, en lugar de limitarse a devolver el mayor de los enteros negativos (es decir, el que tenga un módulo más pequeño) de la entrada. La razón es que la subsecuencia vacía, compuesta de cero enteros, también es una subsecuencia y su suma es claramente 0. Puesto que la subsecuencia vacía es contigua, siempre habrá una subsecuencia contigua cuya suma sea 0. Este resultado es análogo al caso del conjunto vacío, que siempre se considera un subconjunto de cualquier conjunto. Tenga en cuenta que las soluciones vacías siempre son una posibilidad y que en muchas circunstancias no se trata en absoluto de un caso especial.

Hay un montón de algoritmos enormemente diferentes (en términos de eficiencia) que pueden utilizarse para resolver el problema de la suma máxima de una subsecuencia contigua.

El problema de la suma máxima de una subsecuencia contigua es interesante, fundamentalmente, porque existen muchos algoritmos distintos disponibles para resolverlo –y el rendimiento de estos algoritmos varía enormemente. En esta sección vamos a analizar tres de estos algoritmos. El primero es un algoritmo obvio de búsqueda exhaustiva, que resulta muy ineficiente. El segundo, es una mejora del primero, que se consigue realizando una simple observación. El tercero es un algoritmo muy eficiente, aunque no tan obvio. Demostraremos que su tiempo de ejecución es lineal.

En el Capítulo 7 veremos un cuarto algoritmo, que tiene un tiempo de ejecución $O(N \log N)$. Dicho algoritmo no es tan eficiente como el algoritmo lineal, pero es mucho más eficiente que los otros dos. También es un ejemplo típico de los algoritmos con tiempos 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 obvio $O(N^3)$

El algoritmo más simple es una búsqueda exhaustiva directa, o un *algoritmo de fuerza bruta*, como se muestra en la Figura 5.4. Las líneas 9 y 10 controlan un par de bucles que iteran a lo

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

Figura 5.4 Un algoritmo cúbico de suma máxima de subsecuencia contigua.

Un algoritmo de fuerza bruta es, generalmente, el método menos eficiente, pero el más simple de codificar.

largo de todas las posibles subsecuencias. Para cada posible subsecuencia, el valor de su suma se calcula en las líneas 12 a 15. Si esa suma es la mejor de las encontradas hasta el momento, se actualiza el valor de `maxSum`, que se termina devolviendo en la línea 25. También se actualizan dos valores `int`, `seqStart` y `seqEnd` (que son campos de clase estáticos y que indican el principio y el final de la subsecuencia) cada vez que se encuentra una nueva mejor secuencia.

El algoritmo directo de búsqueda exhaustiva tiene como mérito su extremada simplicidad; cuanto menos complejo sea un algoritmo, más probable será que se programe correctamente. Sin embargo, los algoritmos de búsqueda exhaustiva no suelen ser los más eficientes posible. En el resto de esta sección vamos a ver que el tiempo de ejecución de este algoritmo es cúbico. Contaremos el número de veces (como función del tamaño de la entrada) que se evalúan las expresiones de la Figura 5.4. Lo único que necesitamos es un resultado en notación O mayúscula, por lo que una vez que hayamos encontrado un término dominante, podemos ignorar los términos de menor orden y las constantes multiplicativas.

El tiempo de ejecución del algoritmo está dominado completamente por el bucle `for` más interno de las líneas 14 y 15. Hay cuatro expresiones que se ejecutan repetidamente:

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

Se utiliza un análisis matemático para contar el número de veces que se ejecutan ciertas instrucciones.

El número de veces que se ejecuta la expresión 3 hace que sea el término dominante en las cuatro expresiones. Observe que cada inicialización va acompañada por al menos una comprobación. Estamos ignorando las constantes, por lo que podemos despreciar el coste de las inicializaciones; las inicializaciones no pueden ser el coste dominante de un algoritmo. Puesto que la comprobación representada por la expresión 2 solo da un resultado falso una vez por cada bucle, el número de comprobaciones con resultado falso realizado por la expresión 2 es exactamente igual al número de inicializaciones. En consecuencia, no es dominante. El número de comprobaciones con resultado verdadero en la expresión 2, el número de incrementos realizados por la expresión 3 y el número de ajustes de la expresión 4 son idénticos. Por tanto, el número de incrementos (es decir, el número de veces que se ejecuta la línea 15) es una medida dominante del trabajo realizado en el bucle más interno.

El número de veces que se ejecuta la línea 15 es exactamente igual al número de tripletas ordenadas (i, j, k) que satisfacen $1 \leq i \leq k \leq j \leq N$.[2] La razón es que el índice i recorre toda la matriz, mientras que j va de i al final de la matriz y k va de i a j. Una estimación rápida y aproximada es que el número de tripletas es algo inferior a $N \times N \times N$, o N^3, porque i, j y k pueden asumir cada una de ellas uno de N valores posibles. La restricción adicional $i \leq k \leq j$ reduce este número. Un cálculo preciso es algo difícil de obtener y lo realizamos en el Teorema 5.1.

La parte más importante del Teorema 5.1 no es la demostración, sino el resultado. Hay dos formas de evaluar el número de tripletas. Una consiste en evaluar la suma $\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} 1$. Podríamos evaluar esta suma de dentro hacia fuera (véase el Ejercicio 5.11). Pero, en lugar de ello, vamos a emplear una alternativa.

Teorema 5.1

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

Demostración

Coloque las siguientes $N + 2$ bolas en una caja: N bolas numeradas de 1 a N, una bola roja no numerada y una bola azul no numerada. Extraiga tres bolas de la caja. Si se extrae una bola roja, numérela como la más baja de las bolas numeradas extraídas. Si se extrae una bola azul, numérala como la bola más alta de las bolas numeradas extraídas. Observe que si extraemos tanto una bola roja como una azul, entonces el efecto será tener tres bolas con numeración idéntica. Ordene las tres bolas. Cada una de esas ordenaciones corresponde a una tripleta solución de la ecuación del Teorema 5.1. El número de ordenaciones posibles es el número de formas distintas de extraer tres bolas sin sustitución de $N + 2$ bolas. Esto es similar al problema de seleccionar tres puntos de un grupo de N que hemos evaluado en la Sección 5.2, así que se obtiene inmediatamente el resultado indicado.

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

El resultado del Teorema 5.1 es que el bucle `for` más interno representa un tiempo de ejecución cúbico. El trabajo restante del algoritmo no tiene ninguna influencia, porque se realiza, como mucho, una vez por cada iteración del bucle interno. Dicho de otro modo, el coste de las líneas 17 a 22 es irrelevante, porque que esa parte del código se ejecuta únicamente con la misma frecuencia que la inicialización del bucle `for` interno, en lugar de con la misma frecuencia que el cuerpo repetitivo del bucle `for` interno. En consecuencia, el algoritmo es $O(N^3)$.

No necesitamos realizar cálculos precisos para estimar la proporcionalidad O mayúscula. En la mayoría de los casos podemos utilizar la regla simple consistente en multiplicar los tamaños de todos los bucles. Observe que los bucles consecutivos no se mutliplican.

El anterior argumento combinatorio nos permite calcular con precisión el número de iteraciones del bucle interno. Para el cálculo de la proporcionalidad O mayúscula, esto no es realmente necesario; lo único que necesitamos saber es que el término inicial es igual a una cierta constante multiplicada por N^3. Examinando el algoritmo, vemos un bucle que tiene potencialmente un tamaño N dentro de un bucle que tiene potencialmente un tamaño N, dentro de otro bucle que potencialmente también tiene un tamaño N. Esta configuración nos dice que el triple bucle tiene potencialmente $N \times N \times N$ iteraciones. Esta cantidad potencial es solo unas seis veces mayor que el cálculo preciso que hemos hecho de lo que realmente ocurre. Pero las constantes se ignoran de todos modos, así que podemos adoptar la regla general de que, cuando tengamos bucles anidados, debemos 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, la cota superior no representará una sobreestimación excesiva.[3] Por tanto, un programa que tenga tres bucles anidados, cada uno de los cuales se ejecuta secuencialmente a lo largo de una gran parte de una matriz, tiene una gran probabilidad de exhibir un comportamiento $O(N^3)$. Observe que tres bucles consecutivos (no anidados) exhiben un comportamiento lineal; es el anidamiento lo que conduce a la explosión combinatoria. En consecuencia, para mejorar el algoritmo tenemos que eliminar un bucle.

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

Al eliminar un bucle anidado de un algoritmo, generalmente reducimos el tiempo de ejecución.

Cuando podemos eliminar un bucle anidado del algoritmo, generalmente reducimos el tiempo de ejecución. ¿Cómo hacemos para eliminar un bucle? Obviamente, no siempre podremos hacerlo. Sin embargo, el algoritmo anterior tiene muchos cálculos innecesarios. La ineficiencia que el algoritmo mejorado corrige es el cálculo indebidamente costoso incluido en el bucle `for` interno de la Figura 5.4. El algoritmo mejorado hace uso del hecho de que $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$. En otras palabras, suponga que acabamos de calcular la suma para la subsecuencia $i, \ldots, j-1$. Entonces, calcular la suma para la subsecuencia $i, \ldots, j$ no debería exigir mucho esfuerzo, porque lo único que necesitamos es una suma adicional. Sin embargo, en el algoritmo cúbico, esa información se desperdicia. Si usamos esta observación, obtenemos el algoritmo mejorado mostrado en la Figura 5.5. Vemos que hay dos bucles anidados en lugar de tres, y el tiempo de ejecución es $O(N^2)$.

3 El Ejercicio 5.27 ilustra un caso en el que la multiplicación del tamaño de los bucles nos da una sobreestimación de la proporcionalidad O mayúscula.

5.3.3 Un algoritmo lineal

Si eliminamos otro bucle más, tendremos el algoritmo lineal.

Para pasar de un algoritmo cuadrático a un algoritmo lineal, necesitamos eliminar otro bucle más. Sin embargo, a diferencia de la reducción ilustrada en las Figuras 5.4 y 5.5, en las que la eliminación de un bucle fue simple, el deshacernos de un bucle más no es tan sencillo. El problema reside en que el algoritmo cuadrático sigue constituyendo una búsqueda exhaustiva, es decir, estamos comprobando todas las posibles subsecuencias. La única diferencia entre los algoritmos cuadrático y cúbico es que el coste de probar cada subsecuencia sucesiva es una constante en lugar de ser lineal. Puesto que existe un número cuadrático de subsecuencias, la única forma de poder conseguir una cota subcuadrática es encontrar una manera inteligente de no tomar en consideración un gran número de subsecuencias, sin llegar a calcular su suma y sin comprobar si esa suma es un nuevo máximo. En esta sección vamos a ver cómo se puede hacer esto.

El algoritmo tiene su truco. Utiliza una observación inteligente para saltarse rápidamente un gran número de subsecuencias que nunca podrían llegar a ser la mejor.

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

Figura 5.5 Un algoritmo cuadrático de suma máxima de subsecuencia contigua.

En primer lugar, vamos a eliminar un gran número de subsecuencias posibles, evitando tomarlas en consideración. Claramente, la mejor subsecuencia no puede nunca comenzar con un número negativo, por lo que si `a[i]` es negativo podemos saltarnos el bucle interno e incrementar `i`. Dicho de forma más general, la mejor subsecuencia no puede empezar nunca con una subsubsecuencia negativa.

Por tanto, sea $A_{i,j}$ la subsecuencia que abarca los elementos comprendidos entre i y j, y sea $S_{i,j}$ su suma.

Teorema 5.2 Sea $A_{i,j}$ cualquier secuencia con $S_{i,j} < 0$. Si $q > j$, entonces $A_{i,q}$ no es la máxima subsecuencia contigua.

Demostración La suma de los elementos de A entre i y q es la suma de los elementos de A entre i y j más la suma de los elementos de A comprendidos entre $j + 1$ y q. Por tanto, tendremos que $S_{i,q} = S_{i,j} + S_{j+1,q}$. Puesto que $S_{i,j} < 0$, sabemos que $S_{i,j} < S_{j+1,q}$. Por tanto, $A_{i,q}$ no es una subsecuencia contigua máxima.

En las primeras dos líneas de la Figura 5.6 se muestra una ilustración de las sumas generadas por i, j y q. El Teorema 5.2 demuestra que podemos evitar examinar varias subsecuencias incluyendo una prueba adicional: si `thisSum` es menor que 0, podemos saltar (con `break`) del bucle interno en la Figura 5.5. Intuitivamente, si la suma de una subsecuencia es negativa, entonces no puede tomar parte de la subsecuencia contigua máxima. La razón es que podemos obtener una subsecuencia contigua mayor no incluyendo esa subsecuencia con suma negativa. Esta observación no es suficiente, por sí misma, para reducir el tiempo de ejecución por debajo de la cota cuadrática. También se cumple otra observación similar: todas las subsecuencias contiguas que flanqueen a la subsecuencia contigua máxima deben tener sumas negativas (o 0), ya que en caso contrario las incluiríamos en la subsecuencia máxima. Esta observación tampoco nos permite reducir el tiempo de ejecución por debajo de la cota cuadrática. Sin embargo, una tercera observación, ilustrada en la Figura 5.7, sí que nos lo permite, y vamos a formalizarla con el Teorema 5.3.

Teorema 5.3 Para cualquier i, sea $A_{i,j}$ la primera secuencia con $S_{i,j} < 0$. Entonces, para cualquier $i \le p \le j$ y $p \le q$, o bien $A_{p,q}$ no es una subsecuencia contigua máxima o es igual a una subsecuencia contigua máxima que ya hemos visto.

Demostración Si $p = i$, entonces se aplica el Teorema 5.2. En caso contrario, como en el Teorema 5.2, tendremos que $S_{i,q} = S_{i,p-1} + S_{p,q}$. Puesto que j es el menor índice para el que $S_{i,j} < 0$, se deduce que $S_{i,p-1} \ge 0$. Por tanto, $S_{p,q} \le S_{i,q}$. Si $q > j$ (mostrado en el lado izquierdo de la Figura 5.7), entonces el Teorema 5.2 implica que $A_{i,q}$ no es una subsecuencia contigua máxima, por lo que tampoco lo será $A_{p,q}$. En caso contrario, como se muestra en la parte derecha de la Figura 5.7, la subsecuencia $A_{p,q}$ tiene una suma igual, como máximo, a la de la subsecuencia $A_{i,q}$, que ya habremos visto.

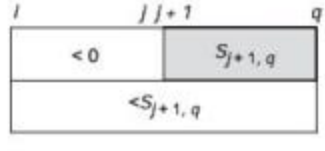

Figura 5.6 Las subsecuencias utilizadas en el Teorema 5.2.

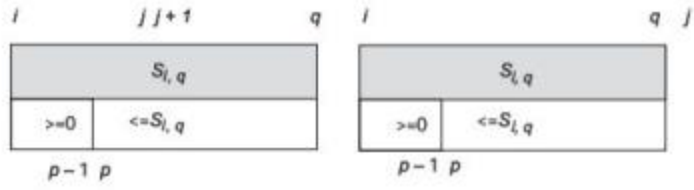

Figura 5.7 Las subsecuencias utilizadas en el Teorema 5.3. La secuencia que va de *p* a *q* tiene una suma que es, como máximo, igual a la de la subsecuencia que va de *i* a *q*. En el lado izquierdo, la secuencia que va de *i* a *q* no es ella misma el máximo (por el Teorema 5.2). En el lado derecho, la secuencia que va de *i* a *q* ya ha sido vista.

El Teorema 5.3 nos dice que, al detectar una subsecuencia negativa, no solo podemos salir con `break` del bucle interno, sino que también podemos incrementar `i` a `j+1`. La Figura 5.8 muestra que podemos reescribir el algoritmo utilizando un único bucle. Claramente, el tiempo de ejecución de

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

Figura 5.8 Un algoritmo lineal de suma máxima de subsecuencia contigua.

Si detectamos una suma negativa, podemos desplazar *i* más allá de *j*.

este algoritmo será lineal: en cada paso del bucle, incrementamos j, por lo que el bucle itera como máximo N veces. La corrección de este algoritmo es mucho menos obvia que para los algoritmos anteriores, lo que es típico de estos procesos de optimización. Es decir, los algoritmos que utilizan la estructura de un problema con el fin de tener mejor rendimiento que una búsqueda exhaustiva requieren, generalmente, algún tipo de prueba de corrección. Hemos demostrado que el algoritmo (aunque no el programa Java resultante) es correcto, utilizando un breve argumento matemático. El propósito es no hacer una exposición completamente matemática, sino más bien dar una ligera pincelada sobre el tipo de técnicas que pueden llegar a ser necesarias a la hora de abordar problemas avanzados.

Si un algoritmo es complejo, hace falta una prueba de corrección.

5.4 Reglas generales para el cálculo de cotas O mayúscula

Ahora que tenemos las ideas básicas acerca del análisis de algoritmos, podemos adoptar un enfoque ligeramente más formal. En esta sección vamos a esbozar las reglas generales para la utilización de la notación O mayúscula. Aunque emplearemos la notación O mayúscula a todo lo largo de este texto, vamos a definir también otros tipos de notaciones algorítmicas que están relacionadas con la notación O mayúscula y que ocasionalmente se usarán más adelante en el libro.

Definición. (O mayúscula) $T(N)$ es $O(F(N))$ si existen sendas constantes positivas c y N_0 tales que $T(N) \leq cF(N)$ cuando $N \geq N_0$.

Definición. (Omega mayúscula) $T(N)$ es $\Omega(F(N))$ si existen sendas constantes positivas c y N_0 tales que $T(N) \geq cF(N)$ cuando $N \geq N_0$.

Definición. (Theta mayúscula) $T(N)$ es $\Theta(F(N))$ si y solo si $T(N)$ es $O(F(N))$ y $T(N)$ es $\Omega(F(N))$.

Definición. (O minúscula) $T(N)$ es $o(F(N))$ si y solo si $T(N)$ es $O(F(N))$ y $T(N)$ no es $\Theta(F(N))$.[4]

La primera definición, notación *O mayúscula*, indica que existe un punto N_0 tal que para todos los valores de N más allá de ese punto, $T(N)$ está acotada por algún múltiplo de $F(N)$. Este es el valor de N suficientemente grande que hemos mencionado anteriormente. Por tanto, si el tiempo de ejecución $T(N)$ de un algoritmo es $O(N^2)$, entonces, ignorando las constantes, estamos garantizando que en algún punto podemos acotar el tiempo de ejecución mediante una función cuadrática. Observe que si el verdadero tiempo de ejecución es lineal, entonces la afirmación de que el tiempo de ejecución es $O(N^2)$ es técnicamente correcta, porque la desigualdad se cumple. Sin embargo, $O(N)$ sería la afirmación más precisa.

[4] Nuestra definición de *O minúscula* no es precisamente correcta para algunas funciones inusuales, pero es la más simple para expresar los conceptos básicos utilizados a través de todo el texto.

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

La notación *O mayúscula* es similar a la desigualdad "menor o igual que", cuando hablamos de tasas de crecimiento.

La segunda definición, $T(N) = \Omega(F(N))$, denominada *Omega mayúscula*, afirma que la tasa de crecimiento de $T(N)$ es mayor o igual que la de $F(N)$. Por ejemplo, podemos decir que cualquier algoritmo que funcione examinando toda posible subsecuencia en el problema de la suma máxima de subsecuencia debe tardar un tiempo $\Omega(N^2)$, porque existe un número cuadrático de subsecuencias posibles. Este es un argumento de cota inferior que se utiliza en análisis más avanzado. Posteriormente en el texto, veremos un ejemplo de este argumento y demostraremos que cualquier algoritmo de ordenación de propósito general requiere un tiempo $\Omega(N \log N)$.

La notación *Omega mayúscula* es similar a la desigualdad "mayor o igual que", cuando hablamos de tasas de crecimiento.

La tercera definición, $T(N) = \Theta(F(N))$, denominada *Theta mayúscula*, dice que la tasa de crecimiento de $T(N)$ es igual a la tasa de crecimiento de $F(N)$. Por ejemplo, el algoritmo de subsecuencia máxima mostrado en la Figura 5.5 se ejecuta en un tiempo $\Theta(N^2)$. En otras palabras, el tiempo de ejecución está acotado por una función cuadrática y esta cota no puede mejorarse porque también posee como cota inferior otra función cuadrática. Cuando utilizamos la notación Theta mayúscula, no estamos solo proporcionando una cota superior de un algoritmo, sino también una garantía de que el análisis que conduce a la determinación de esa cota superior es lo mejor (lo más exacto) posible. Sin embargo, a pesar de la precisión adicional ofrecida por Theta mayúscula, se suele utilizar más comúnmente O mayúscula, excepto por parte de los que investigan en el campo de análisis de algoritmos.

La notación *Theta mayúscula* es similar a la igualdad, cuando hablamos de tasas de crecimiento.

La última definición, $T(N) = o(F(N))$, denominada *O minúscula*, dice que la tasa de crecimiento de $T(N)$ es estrictamente inferior a la tasa de crecimiento de $F(N)$. Esta función es diferente de O mayúscula porque O mayúscula admite la posibilidad de que las tasas de crecimiento coincidan. Por ejemplo, si el tiempo de ejecución de un algoritmo es $o(N^2)$, entonces se garantiza que crece a una tasa más lenta que la cuadrática (es decir, se trata de un *algoritmo subcuadrático*). Por tanto, una cota de $o(N^2)$ es una cota mejor que $\Theta(N^2)$. La Figura 5.9 resume estas cuatro definiciones.

La notación *O minúscula* es similar a la desigualdad "menor que", cuando hablamos de tasas de crecimiento.

Es conveniente proporcionar un par de notas estilísticas. En primer lugar, no constituye un buen estilo incluir constantes o términos de orden inferior dentro de una expresión O mayúscula. No escriba $T(N) = O(2N^2)$ ni $T(N) = O(N^2 + N)$. En ambos casos, la forma correcta es $T(N) = O(N^2)$. En segundo lugar, en cualquier análisis que requiera una respuesta O mayúscula, pueden utilizarse todos los tipos de atajos imaginables. Los términos de menor orden, las constantes multiplicativas y los símbolos relacionales se eliminan siempre.

Elimine las constantes multiplicativas, los términos de orden inferior y los símbolos relacionales cuando utilice la notación O mayúscula.

Ahora que hemos formalizado los aspectos matemáticos, podemos ponerlos en relación con el análisis de algoritmos. La regla más básica es que *el tiempo de ejecución de un bucle es, como máximo, el tiempo de ejecución de las instrucciones contenidas dentro del bucle (incluyendo las comprobaciones) multiplicado por el número de iteraciones*. Como hemos visto anteriormente, la inicialización y prueba de la condición del bucle no suele ser más dominante que las instrucciones que componen el cuerpo del bucle.

Expresión matemática	Tasas relativas de crecimiento
$T(N) = O(F(N))$	Crecimiento de $T(N) \leq$ crecimiento de $F(N)$.
$T(N) = \Omega(F(N))$	Crecimiento de $T(N) \geq$ crecimiento de $F(N)$.
$T(N) = \Theta(F(N))$	Crecimiento de $T(N) =$ crecimiento de $F(N)$.
$T(N) = o(F(N))$	Crecimiento de $T(N) <$ crecimiento de $F(N)$.

Figura 5.9 Significado de las distintas funciones de crecimiento.

Una *cota de caso peor* proporciona una garantía para todas las entradas de un cierto tamaño.

El tiempo de ejecución de las instrucciones contenidas dentro de un grupo de bucles anidados es igual al tiempo de ejecución de las instrucciones (incluyendo las comprobaciones del bucle más interno) multiplicado por los tamaños de todos los bucles. El tiempo de ejecución de una secuencia de bucles consecutivos es igual al tiempo de ejecución de bucle dominante. La diferencia de tiempo entre un bucle anidado en el que ambos índices vayan de 1 a N y dos bucles consecutivos que no estén anidados, pero que se ejecuten a lo largo del mismo rango de índices, es similar a la diferencia de espacio existente entre una matriz bidimensional y dos matrices unidimensionales. El primer caso es cuadrático. El segundo caso es lineal, porque $N + N$ es $2N$, que sigue siendo $O(N)$. En ocasiones, esta regla simple puede sobreestimar el tiempo de ejecución, pero en la mayoría de los casos no lo hace. Incluso si lo hace, O mayúscula no garantiza una respuesta asintótica exacta, simplemente proporciona una cota superior.

Los análisis que hemos realizado hasta ahora implicaban utilizar una *cota de caso peor*, que proporciona una garantía para todas las entradas de un cierto tamaño. Otra forma de análisis es el de *cota de caso promedio*, en el que el tiempo de ejecución se mide como un promedio para todas las posibles entradas de tamaño N. El promedio puede diferir del caso peor si, por ejemplo, una instrucción condicional que depende de la entrada concreta provoca una salida anticipada de un bucle. Hablaremos con más detalle de las cotas de caso promedio en la Sección 5.8. Por ahora, observe simplemente que el hecho de que un algoritmo tenga una cota de caso peor mejor que otro algoritmo no nos dice nada acerca de cómo se comparan sus cotas de caso promedio respectivas. Sin embargo, en muchos casos, las cotas de caso promedio y de caso peor están estrechamente correlacionadas. Cuando no lo están, esas cotas se analizan por separado.

En una *cota de caso promedio*, el tiempo de ejecución se mide como promedio para todas las posibles entradas de tamaño N.

El último aspecto del análisis O mayúscula que vamos a examinar es el de cómo crece el tiempo de ejecución para cada tipo de curva, como se ilustra en las Figuras 5.1 y 5.2. Lo que queremos es una respuesta más cuantitativa a esta cuestión: si un algoritmo tarda $T(N)$ en resolver un problema de tamaño N, ¿cuánto tardará en resolver un problema de mayor tamaño? Por ejemplo, ¿cuánta tarda en resolver un problema cuando la entrada es 10 veces mayor? Las respuestas a estas cuestiones se muestran en la Figura 5.10. Sin embargo, queremos responder a esta cuestión sin ejecutar el programa y esperamos que nuestras respuestas analíticas concuerden con el comportamiento observado.

	Figura 5.4	Figura 5.5	Figura 7.20	Figura 5.8
N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0,000001	0,000000	0,000001	0,000000
100	0,000288	0,000019	0,000014	0,000005
1.000	0,223111	0,001630	0,000154	0,000053
10.000	218	0,133064	0,001630	0,000533
100.000	No disponible	13,17	0,017467	0,005571
1.000.000	No disponible	No disponible	0,185363	0,056338

Figura 5.10 Tiempos de ejecución (en segundos) observados para varios algoritmos de suma máxima de subsecuencia contigua.

Comencemos examinando el algoritmo cúbico. Asumimos que el tiempo de ejecución está razonablemente aproximado por $T(N) = cN^3$. En consecuencia, $T(10\ N) = c(10\ N)^3$. Una manipulación matemática nos da

$$T(10N) = 1000cN^3 = 1000T(N)$$

Si el tamaño de la entrada se multiplica por un factor f, el tiempo de ejecución de un programa cúbico se multiplica por un factor aproximadamente igual a f^3.

Por tanto, el tiempo de ejecución de un programa cúbico se multiplica por un factor de 1.000 (asumiendo que N sea suficientemente grande) cuando la cantidad de entrada se multiplica por un factor de 10. Esta relación se ve confirmada aproximadamente por el incremento en tiempo de ejecución entre $N = 100$ y 1.000 mostrado en la Figura 5.10. Recuerde que no esperamos obtener una respuesta exacta –simplemente una aproximación razonable. También esperaríamos que para $N = 100.000$, el tiempo de ejecución se multiplicara de nuevo por 1.000. El resultado sería que un algoritmo cúbico requeriría aproximadamente 60 horas (2 días y medio) de tiempo de computación. En general, si la cantidad de entrada se multiplica por un factor f, entonces el tiempo de ejecución del algoritmo cúbico se multiplica por un factor f^3.

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

Podemos realizar cálculos similares para los algoritmos cuadrático y lineal. Para el algoritmo cuadrático, suponemos que $T(N) = cN^2$. De aquí se deduce que $T(10\ N) = c(10N)^2$. Al expandir tenemos

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

Por tanto, cuando el tamaño de la entrada se multiplica por un factor de 10, el tiempo de ejecución de un programa cuadrático se multiplica por un factor de aproximadamente 100. Esta relación se ve también confirmada en la Figura 5.10. En general, para un algoritmo cuadrático, una multiplicación por un factor f en el tamaño de la entrada nos da una multiplicación por f^2 en el tiempo de ejecución.

Finalmente, para un algoritmo lineal, un cálculo similar nos muestra que si multiplicamos el tamaño de la entrada por 10, obtenemos un tiempo de ejecución que también se multiplica por 10. De nuevo, esta relación se ve confirmada experimentalmente en la Figura 5.10. Observe,

Si el tamaño de la entrada se multiplica por un factor *f*, el tiempo de ejecución de un programa lineal también se multiplica por un factor *f*. Este es el tiempo de ejecución preferido para un algoritmo.

sin embargo, que para un programa lineal, el término *suficientemente grande* podría significar un tamaño de entrada algo mayor que para los restantes programas, suponiendo que en todos los casos existe un tiempo de computación mínimo fijo que es independiente del tamaño de la entrada. Para un programa lineal, este término podría continuar teniendo un valor significativo para tamaños de entrada moderados.

El análisis utilizado aquí no funciona cuando hay términos logarítmicos. Cuando a un algoritmo $O(N \log N)$ se le presenta una entrada 10 veces mayor, el tiempo de ejecución se multiplica por un factor ligeramente mayor de 10. Específicamente, tenemos que $T(10\,N) = c(10N) \log (10\,N)$. Al expandir, obtenemos

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

Aquí $c' = 10\,c \log 10$. A medida que N va haciéndose muy grande, la relación $T(10\,N) / T(N)$ va haciéndose más y más próxima a 10, porque $c'\,N / T(N) \approx (10 \log 10)/\log N$ se hace cada vez más pequeño a medida que aumenta N. En consecuencia, si el algoritmo es competitivo comparado con un algoritmo lineal para un valor de N muy grande, es probable que siga siendo competitivo para un valor de N ligeramente mayor.

¿Quiere esto decir que los algoritmos cuadráticos y cúbicos son inútiles? La respuesta es no. En algunos casos, los algoritmos más eficientes conocidos son cuadráticos o cúbicos. En otros, el algoritmo más eficiente es aún peor (exponencial). Además, cuando la cantidad de entrada es pequeña, cualquier algoritmo nos sirve. Frecuentemente, los algoritmos que no son asintóticamente eficientes, resultan ser, de todos modos, fáciles de programar. Para tamaños de entrada pequeños, esa es la manera correcta de proceder. Finalmente, una buena forma de probar un algoritmo lineal complejo consiste en comparar su salida con la de un algoritmo de búsqueda exhaustiva. En la Sección 5.8 hablaremos, asimismo, de algunas otras limitaciones del modelo O mayúscula.

5.5 El logaritmo

La lista de funciones de crecimiento típicas incluye varias entradas que contienen el logaritmo. Un *logaritmo* es el exponente que indica la potencia a la que hay que elevar un número (la base) para obtener otro número dado. En esta sección vamos a ver con algo más de detalle las matemáticas que subyacen al concepto de logaritmo. En la Sección 5.6 veremos cómo se utilizan en un algoritmo sencillo.

Comenzaremos con la definición formal y luego aportaremos algunos puntos de vista más intuitivos.

El *logaritmo* de *N* (en base 2) es el valor *X* tal que 2 elevado a la potencia *X* es igual a *N*. De manera predeterminada, la base del algoritmo es 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 informática, cuando se omite la base, se asume de manera predeterminada que es 2, lo cual es natural por diversas razones, como veremos posteriormente en el capítulo. Vamos a demostrar un teorema matemático, el Teorema 5.4 para ver que, en lo que respecta a la notación O mayúscula, la base no tiene ninguna importancia, y también para mostrar cómo deducir las relaciones en las que están implicados logaritmos.

Teorema 5.4 La base no importa. 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 $2^C = B$. Por tanto, $B^K = (2^C)^K = N$. De aquí, tenemos que $2^{CK} = N$, lo que 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 de este texto, utilizaremos exclusivamente logaritmos en base 2. Un hecho importante acerca del logaritmo es que es una función que crece lentamente. Puesto que $2^{10} = 1.024$, $\log 1.024 = 10$. Cálculos adicionales nos muestran que el logaritmo de 1.000.000 es aproximadamente 20 y el logaritmo de 1.000.000.000 es solo 30. En consecuencia, el rendimiento de un algoritmo $O(N \log N)$ está mucho más próximo a un algoritmo lineal $O(N)$ que a uno cuadrático $O(N^2)$, incluso para cantidades de entrada moderadamente grandes. Antes de examinar un algoritmo realista cuyo tiempo de ejecución incluye el logaritmo, veamos unos cuantos ejemplos del papel que desempeñan los logaritmos.

Bits en un número binario

¿Cuántos bits hacen falta para representar N enteros consecutivos?

El número de bits requerido para representar números es logarítmico.

Un entero `short` de 16 bits representa 65.536 enteros comprendidos en el rango que va de −32.768 a 32.767. En general, bastan B bits para representar 2^B enteros distintos. Por tanto, el número B de bits requeridos para representar N enteros consecutivos satisface la ecuación $2^B \geq N$. De aquí, obtenemos que $B \geq \log N$, por lo que el número mínimo de bits es $\lceil \log N \rceil$. (Aquí $\lceil X \rceil$ es la función techo y representa el entero más pequeño que tiene al menos un valor igual a X. La correspondiente función suelo $\lfloor X \rfloor$ representa el entero que tiene un valor al menos tan pequeño como X.)

Duplicaciones consecutivas

Comenzando con $X = 1$, ¿cuántas veces hay que multiplicar X por dos antes de que tenga un valor al menos tan grande como el de N?

El *principio de la duplicación repetida* afirma que, partiendo de 1 solo podemos multiplicar por dos repetidamente un número logarítmico de veces, hasta alcanzar N.

Suponga que comenzamos 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 tendríamos 4 euros; después de 3 años, tendríamos 8 euros, y así sucesivamente. En general, después de K años tendríamos 2^K euros, por lo que queremos encontrar el valor de K más pequeño que satisfaga $2^K \geq N$. Esta es la misma ecuación que antes, por lo que $K = \lceil \log N \rceil$. Después de 20 años, tendríamos más de un millón de euros. El *principio de la duplicación repetida* afirma que, partiendo de 1, solo podemos duplicar la cantidad repetidamente $\lceil \log N \rceil$ veces hasta alcanzar N.

Divisiones consecutivas

Comenzando con $X = N$, si dividimos N entre dos repetidamente, ¿cuántas iteraciones habrá que aplicar para que N sea menor o igual que 1?

Si se redondea el resultado de la división al entero más próximo (o si efectuamos una división real y no entera), tenemos el mismo problema que con las duplicaciones repetidas, salvo porque ahora

El *principio de la división repetida* afirma que, partiendo de N, solo podemos dividir entre dos un número logarítmico de veces. Este proceso se utiliza para obtener rutinas de búsqueda logarítmicas.

vamos en la dirección opuesta. De nuevo, la respuesta es $\lceil \log N \rceil$ iteraciones. Si se redondea hacia abajo la división, la respuesta es $\lfloor \log N \rfloor$. Podemos ver la diferencia comenzando con $X = 3$. Hacen falta dos divisiones, a menos que redondeemos las divisiones por defecto, en cuyo caso solo hace falta una.

Muchos de los algoritmos examinados en este texto tendrán logaritmos, introducidos a causa del *principio de la división repetida* que afirma que, si comenzamos con N, solo podemos dividir entre dos un número logarítmico de veces. En otras palabras, un algoritmo es $O(N \log N)$ si hace falta un tiempo constante ($O(1)$) para reducir el tamaño del problema según una fracción constante (que normalmente es 1/2). Esta condición se deduce directamente del hecho de que habrá $O(\log N)$ iteraciones del bucle. Cualquier fracción constante nos sirve, porque la fracción se ve reflejada en la base del logaritmo, y el Teorema 5.4 nos dice que la base no importa.

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

Todas las restantes apariciones de los logaritmos se deben (directa o indirectamente) a la aplicación del Teorema 5.5. Este teorema está relacionado con el N-ésimo número armónico, que es la suma de los recíprocos de los primeros N enteros positivos, y afirma que el N-ésimo número armónico, H_N, satisface $H_N = \Theta(\log N)$. La demostración utiliza conceptos de calculo, pero no hace falta entender la demostración para utilizar el teorema.

Teorema 5.5

Sea $H_N = \sum_{i=1}^{N} 1/i$. Entonces $H_N = \Theta(\log N)$. Una estimación más precisa es $\ln N + 0{,}577$.

Demostración

La intuición de la demostración es que una suma discreta está bien aproximada por la integral (continua). La demostración utiliza una construcción para demostrar que la suma H_N puede acotarse por arriba y por abajo mediante $\int \frac{dx}{x}$, con límites apropiados. Los detalles se dejan para el Ejercicio 5.28.

En la siguiente sección se muestra cómo el principio de la división repetida conduce a un algoritmo de búsqueda eficiente.

5.6 Problema de la búsqueda estática

Uno de los usos más importantes de las computadoras es el de la búsqueda de datos. Si no se permite que los datos varíen (por ejemplo, si están almacenados en CD-ROM), decimos que los datos son estáticos. Una *búsqueda estática* accede a datos que nunca son modificados. El problema de la búsqueda estática se puede formular de manera natural de la forma siguiente:

Problema de la búsqueda estática

Dado un entero X y una matriz A, devolver la posición de X en A o una indicación de que no está presente. Si X aparece más de una vez, devolver cualquiera de las apariciones. La matriz A nunca es modificada.

Un ejemplo de búsqueda estática sería buscar una persona en una guía telefónica. La eficiencia de un algoritmo de búsqueda estática depende de si la matriz en la que se está buscando está ordenada. En el caso de la guía telefónica, buscar por nombre es muy rápido, pero buscar por