7.5 Algoritmos de tipo divide y vencerás

Un *algoritmo de tipo divide y vencerás* es un algoritmo recursivo que, por regla general, es muy eficiente.

Una importante técnica de resolución de problemas que hace uso de la recursión es la conocida como técnica de divide y vencerás. Un *algoritmo de divide y vencerás* es un eficiente algoritmo recursivo que está compuesto de dos partes:

- *Divide*, durante la cual se resuelven recursivamente problemas sucesivamente más pequeños (excepto, por supuesto, los casos base, que se resuelven directamente).
- *Vencerás*, en la que se forma la solución al problema original componiendo las soluciones a los subproblemas.

En la técnica de *divide y vencerás*, la recursión es el *divide* y el procesamiento adicional es el *vencerás*.

Tradicionalmente, las rutinas en las que el algoritmo contiene al menos dos llamadas recursivas se denominan algoritmos de tipo divide y vencerás, mientras que las rutinas cuyo texto contiene solo una llamada recursiva no se clasifican dentro de esa categoría. En consecuencia, las rutinas recursivas presentadas hasta ahora en este capítulo no son algoritmos de tipo divide y vencerás. Asimismo, los subproblemas usualmente deben ser disjuntos (es decir, esencialmente no solapados), para evitar los excesos de costes que ya hemos podido ver en los cálculos recursivos de ejemplo de los números de Fibonacci.

En esta sección proporcionamos un ejemplo del paradigma divide y vencerás. En primer lugar, veremos cómo utilizar la recursión para resolver el problema de la suma máxima de subsecuencia. Después, proporcionaremos un análisis para demostrar que el tiempo de ejecución es $O(N \log N)$. Aunque ya hemos utilizado un algoritmo lineal para este problema, la solución es bastante ilustrativa de un conjunto de soluciones que tienen un amplio rango de aplicaciones, incluyendo los algoritmos de ordenación, como el de ordenación por mezcla y la ordenación rápida, que veremos en el Capítulo 8. En consecuencia, aprender la técnica es importante. Finalmente, mostraremos la forma general para el cálculo del tiempo de ejecución de una amplia gama de algoritmos de tipo divide y vencerás.

7.5.1 El problema de la suma máxima de subsecuencia contigua

En la Sección 5.3 hemos hablado del problema de encontrar, dentro de una secuencia de números, una subsecuencia contigua cuya suma sea máxima. Por comodidad, vamos a volver a enunciar el problema aquí.

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.

Allí presentamos tres algoritmos de diferente complejidad. Uno era un algoritmo basado en una búsqueda exhaustiva: calculábamos la suma de cada subsecuencia posible y luego seleccionábamos el máximo. También describimos una mejora cuadrática que aprovechaba el hecho de que cada subsecuencia puede calcularse en un tiempo constante a partir de una subsecuencia anterior. Puesto

El problema de la suma máxima de una subsecuencia contigua puede resolver con un algoritmo de tipo divide y vencerás.

que tenemos $O(N^2)$ subsecuencias, esta cota es la mejor que se puede obtener cuando se emplea una técnica en la que se examinan directamente todas las subsecuencias. También proporcionamos un algoritmo de tiempo lineal que examinaba solo unas pocas subsecuencias. Sin embargo, la corrección de ese algoritmo no era obvia.

Vamos a considerar un algoritmo de tipo divide y vencerás. Suponga que la secuencia de entrada de ejemplo es $\{4, -3, 5, -2, -1, 2, 6, -2\}$. Vamos a dividir esta entrada en dos mitades como se muestra en la Figura 7.19. Entonces, la suma máxima de subsecuencia contigua puede producirse de tres formas distintas:

- *Caso* 1: reside enteramente en la primera mitad.
- *Caso* 2: reside enteramente en la segunda mitad.
- *Caso* 3: comienza en la primera mitad y termina en la segunda mitad.

Vamos a ver cómo encontrar los máximos de cada uno de estos tres casos de manera más eficiente que utilizando una búsqueda exhaustiva.

Comencemos examinando el caso 3. Queremos evitar el bucle anidado que resultaría de considerar todos los $N/2$ puntos iniciales y los $N/2$ puntos finales independientemente. Podemos evitar esa situación sustituyendo dos bucles anidados por dos bucles consecutivos. Los bucles consecutivos, cada uno de ellos de tamaño $N/2$, se combinan para dar lugar a un procesamiento de carácter simplemente lineal. Podemos hacer esta sustitución porque cualquier subsecuencia contigua que comience en la primera mitad y termine en la segunda mitad debe incluir tanto el último elemento de la primera mitad como el primer elemento de la segunda mitad.

La Figura 7.19 muestra que, para cada elemento de la primera mitad, podemos calcular la suma de subsecuencia contigua que termina en el elemento situado más a la derecha. Podemos hacer esto con una exploración que vaya de derecha a izquierda, comenzando por la línea divisoria entre las dos mitades. De forma similar, podemos calcular la suma de subsecuencia contigua para todas las subsecuencias que comiencen con el primer elemento de la segunda mitad. Después, podemos combinar estas dos subsecuencias para formar la máxima subsecuencia contigua que abarque la frontera divisoria. En este ejemplo, la secuencia resultante va desde el primer elemento de la primera mitad al penúltimo elemento de la segunda mitad. La suma total es la suma de las dos subsecuencias, es decir $4 + 7 = 11$.

Este análisis muestra que el caso 3 se puede resolver en un tiempo lineal. ¿Pero qué sucede con los casos 1 y 2? Puesto que hay $N/2$ elementos en cada mitad, una búsqueda exhaustiva aplicada a cada mitad seguirá requiriendo un tiempo cuadrático; específicamente, lo único que habremos hecho es eliminar la mitad del trabajo, pero la mitad de un tiempo cuadrático sigue siendo cuadrático.

Primera mitad				Segunda mitad				
4	–3	5	–2	–1	2	6	–2	Valores
4*	0	3	–2	–1	1	7*	5	Suma acumulada

Suma acumulada a partir del centro (*indica el máximo de cada una de las mitades).

Figura 7.19 División del problema de la subsecuencia máxima contigua en dos mitades.

En consecuencia, en los casos 1 y 2 lo que hacemos es aplicar la misma estrategia –dividir en más mitades. Podemos continuar dividiendo esas mitades una y otra vez hasta que la división sea imposible. Este enfoque se puede enunciar sucintamente de la forma siguiente: *resolver los casos 1 y 2 recursivamente*. Como demostraremos más adelante, al hacer esto se reduce el tiempo de ejecución por debajo de la cota cuadrática, porque los ahorros se van acumulando a lo largo del algoritmo. A continuación se muestra un resumen de la parte principal del algoritmo.

1. Calcular recursivamente la suma máxima de las subsecuencias contiguas que residan enteramente en la primera mitad.
2. Calcular recursivamente la suma máxima de las subsecuencias contiguas que residan enteramente en la segunda mitad.
3. Calcular, mediante dos bucles consecutivos, la suma máxima de las subsecuencias contiguas que comiencen en la primera mitad y terminen en la segunda mitad.
4. Seleccionar el máximo de las tres sumas.

Todo algoritmo recursivo requiere que se especifique un caso base. Cuando el tamaño del problema sea de solo un elemento, no utilizaremos recursión. El método Java resultante se muestra en la Figura 7.20.

La forma general de la llamada recursiva consiste en pasar la matriz de entrada junto con los límites derecho e izquierdo, que delimitan la parte de la matriz sobre la que se está operando. Una rutina de preparación de una única línea se encarga de preparar el terreno, pasando como parámetros los límites 0 y $N-1$ junto con la matriz.

Las líneas 12 y 13 se encargan del caso base. Si `left==right`, habrá un solo elemento, y será la subsecuencia máxima contigua en caso de que el elemento sea no negativo (en caso contrario, la secuencia vacía de suma 0 será la máxima). Las líneas 15 y 16 realizan las dos llamadas recursivas. Estas llamadas siempre se aplican a un problema más pequeño que el problema original; por tanto, vamos progresando hacia el caso base. Las líneas 18 a 23 y 25 a 30 calculan las sumas máximas de las secuencias que comienzan en el borde central. La suma de estos dos valores será la suma máxima de las secuencias que abarquen ambas mitades. La rutina `max3` (no mostrada aquí) devuelve el valor máximo de las tres posibilidades disponibles.

7.5.2 Análisis de una recurrencia básica de tipo divide y vencerás

Análisis intuitivo del algoritmo de tipo divide y vencerás para el cálculo de la suma máxima de subsecuencia contigua: invertimos $O(N)$ por cada nivel.

El algoritmo recursivo de cálculo de la suma máxima de subsecuencia contigua funciona realizando un procesamiento lineal para calcular una suma que incluye el borde central, y realizando luego dos llamadas recursivas. Estas llamadas, en conjunto, calculan una suma que incluye el borde central, efectúan llamadas recursivas adicionales, etc. El trabajo total realizado por el algoritmo será entonces proporcional al trabajo de exploración realizado en todas las llamadas recursivas.

La Figura 7.21 ilustra gráficamente cómo funciona el algoritmo para $N = 8$ elementos. Cada rectángulo representa una llamada a `maxSumRec` y la longitud del rectángulo es proporcional al tamaño de la submatriz (y por tanto al coste de explorar la submatriz con la que se está operando en

```
1   /**
2    * Algoritmo recursivo de suma máxima de subsecuencia contigua.
3    * Encuentra la suma máxima en una submatriz que abarca a[left..right].
4    * No trata de mantener la mejor secuencia actual.
5    */
6   private static int maxSumRec( int [ ] a, int left, int right )
7   {
8      int maxLeftBorderSum = 0, maxRightBorderSum = 0;
9      int leftBorderSum = 0, rightBorderSum = 0;
10     int center = ( left + right ) / 2;
11
12     if( left == right ) // Caso base
13        return a[ left ] > 0 ? a[ left ] : 0;
14
15     int maxLeftSum = maxSumRec( a, left, center );
16     int maxRightSum = maxSumRec( a, center + 1, right );
17
18     for( int i = center; i >= left; i-- )
19     {
20        leftBorderSum += a[ i ];
21        if( leftBorderSum > maxLeftBorderSum )
22        maxLeftBorderSum = leftBorderSum;
23     }
24
25     for( int i = center + 1; i <= right; i++ )
26     {
27        rightBorderSum += a[ i ];
28        if( rightBorderSum > maxRightBorderSum )
29           maxRightBorderSum = rightBorderSum;
30     }
31
32        return max3( maxLeftSum, maxRightSum,
33                     maxLeftBorderSum + maxRightBorderSum );
34  }
35
36  /**
37   * Rutina de preparación para el algoritmo de tipo divide y vencerás
38   * para el cálculo de la suma máxima de subsecuencia contigua.
39   */
40  public static int maxSubsequenceSum( int [ ] a )
41  {
42     return a.length > 0 ? maxSumRec( a, 0, a.length - 1 ) : 0;
43  }
```

Figura 7.20 Un algoritmo de tipo divide y vencerás para el problema del cálculo de la suma máxima de subsecuencia contigua.

esa invocación). La llamada inicial se muestra en la primera línea: el tamaño de la submatriz es N, que representa el coste de exploración asociado con el tercer caso. La llamada inicial hace entonces dos llamadas recursivas, lo que nos da dos submatrices de tamaño $N/2$. El coste de cada exploración en el caso 3 será la mitad del coste original, pero como hay dos llamadas recursivas, el coste combinado de dichas llamadas será también N. Cada una de estas dos instancias recursivas hace a su vez otras dos llamadas recursivas, lo que nos da cuatro subproblemas que tienen un tamaño igual a un cuarto del tamaño original. Por tanto, el total de los costes asociados con el caso 3 es también N.

Al final, terminaremos por llegar al caso base. Cada caso base tiene un tamaño igual a 1 y hay N de esos casos. Por supuesto, en este caso no hay ningún coste asociado con el caso 3, pero calculamos una unidad de coste por realizar la comprobación que determina si el único elemento restante es positivo o negativo. El coste total es, por tanto, como se ilustra en la Figura 7.21, N por cada nivel de recursión. Cada nivel divide a la mitad el tamaño del problema básico, por lo que el principio de división por la mitad nos dice que habrá aproximadamente log N niveles. De hecho, el número de niveles es $1 + \lceil \log N \rceil$ (que será igual a 4 cuando N sea igual a 8). Por tanto, cabe esperar que el tiempo total de ejecución sea $O(N \log N)$.

Este análisis nos da una explicación intuitiva de por qué el tiempo de ejecución es $O(N \log N)$. Sin embargo, en general, expandir un algoritmo recursivo para examinar su comportamiento no es buena idea; viola la tercera regla de la recursión. A continuación, vamos a considerar un tratamiento matemático más formal.

Observe que el análisis más formal es aplicable a todas las clases de algoritmos que resuelvan recursivamente dos mitades y utilicen un trabajo adicional lineal.

Sea $T(N)$ el tiempo requerido para resolver un problema de suma máxima de subsecuencia contigua de tamaño N. Si $N = 1$, el programa requiere una cierta cantidad constante de tiempo para ejecutar las líneas 12 a 13, la cual consideraremos que es igual a 1 unidad. Por tanto, $T(1) = 1$. En caso contrario, el programa debe efectuar dos llamadas recursivas y el trabajo lineal implicado en calcular la suma máxima para el caso 3. El procesamiento constante adicional es absorbido por el término $O(N)$. ¿Cuánto tardan las dos llamadas recursivas? Puesto que resuelve problemas de tamaño $N/2$, sabemos que cada una debe requerir $T(N/2)$ unidades de tiempo; en consecuencia, el trabajo recursivo total es $2T(N/2)$. Este análisis nos da las ecuaciones

$$T(1) = 1$$
$$T(N) = 2T(N/2) + O(N)$$

Por supuesto, para que la segunda ecuación tenga sentido, N debe ser una potencia de 2. En caso contrario, en algún punto $N/2$ no sería par. Una ecuación más precisa sería

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N)$$

Para simplificar los cálculos, vamos a asumir que N es una potencia de 2 y a sustituir el término $O(N)$ por N. Estas suposiciones no tienen demasiada importancia y no afectan al resultado O mayúscula. En consecuencia, necesitamos obtener una solución explícita para $T(N)$ a partir de

$$T(1) = 1 \quad \text{y} \quad T(N) = 2T(N/2) + N \tag{7.6}$$

Esta ecuación se ilustra en la Figura 7.21, así que sabemos que la respuesta será $N \log N + N$. Podemos verificar fácilmente el resultado examinando unos cuantos valores: $T(1)$, $T(2) = 4$, $T(4) = 12$, $T(8) = 32$ y $T(16) = 80$. Ahora vamos a demostrar este análisis matemáticamente en el Teorema 7.4, utilizando dos método diferentes.

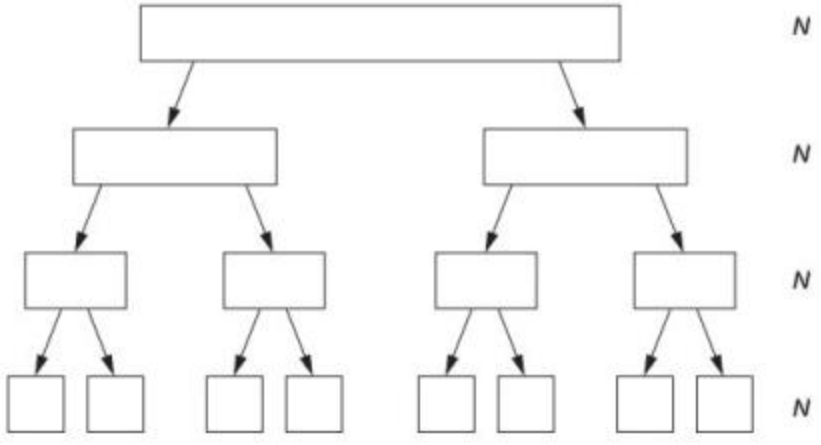

Figura 7.21 Traza de las llamadas recursivas en el algoritmo recursivo de cálculo de la suma máxima de subsecuencia contigua, en el caso de $N = 8$ elementos.

Teorema 7.4

Suponiendo que N es una potencia de 2, la solución a la ecuación $T(N) = 2T(N/2) + N$ con la condición inicial $T(1) = 1$, es $T(N) = N\log N + N$.

Demostración (método 1)

Para un valor de N suficientemente grande, tenemos que $T(N/2) = 2T(N/4) + N/2$, porque podemos utilizar la Ecuación 7.6 con $N/2$ en lugar de N. En consecuencia, tenemos

$$2T(N/2) = 4T(N/4) + N$$

Sustituyendo esto en la Ecuación 7.6, obtenemos

$$T(N) = 4T(N/4) + 2N \qquad \textbf{(7.7)}$$

Si empleamos la Ecuación 7.6 para $N/4$ y multiplicamos por 4, obtenemos

$$4T(N/4) = 8T(N/8) + N$$

que podemos utilizar para sustituirlo en el lado derecho de la Ecuación 7.7, obteniendo

$$T(N) = 8T(N/8) + 3N$$

Continuando de esta forma, obtenemos

$$T(N) = 2^k T(N/2^k) + kN$$

Finalmente, utilizando $k = \log N$ (lo que tiene sentido, porque entonces $2^k = N$), obtenemos

$$T(N) = NT(1) + N\log N = N\log N + N$$

Aunque este método de demostración parece funcionar bien, puede ser difícil de aplicar en algunos otros casos más complicados, porque tiende a dar ecuaciones muy largas. A continuación se muestra un segundo método que parece ser más fácil, porque genera ecuaciones verticalmente que resultan más fáciles de manipular.

Demostración del Teorema 7.4 (método 2)

Dividimos la Ecuación 7.6 entre N, lo que nos una nueva ecuación:

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

Esta ecuación ahora es válida para cualquier N que sea una potencia de 2, por lo que podemos escribir las siguientes ecuaciones

$$\begin{aligned}
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + 1 \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + 1 \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + 1 \\
&\cdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + 1
\end{aligned} \tag{7.8}$$

Una *suma telescópica* genera un gran número de términos que se cancelan.

Ahora sumamos todas las ecuaciones de la Ecuación 7.8. Es decir, sumamos todos los términos situados en el lado izquierdo e igualamos el resultado a la suma de todos los términos del lado derecho. El término $T(N/2) / (N/2)$ aparece en ambos lados y por tanto se cancela. De hecho, casi todos los términos aparecen en ambos lados y se cancelan. Esto es lo que se denomina *suma telescópica*. Después de sumarlo todo, el resultado final es

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

porque todos los demás términos se cancelan y hay $\log N$ ecuaciones, por lo que todos los 1 que aparecen en estas ecuaciones suman $\log N$. Multiplicando por N obtenemos la respuesta final que coincide con la obtenida anteriormente.

Observe que, si no hubiéramos dividido por N al principio de la solución, no se podría haber formado la suma telescópica. Decidir cuál es la división necesaria para garantizar la obtención de una suma telescópica requiere algo de experiencia, y hace que el método sea un poco más difícil de aplicar que la primera alternativa. Sin embargo, una vez que se ha encontrado el divisor correcto, la segunda alternativa tiende a producir una serie de cálculos que encajan mejor en una hoja de papel, lo que conduce a que se produzca un menor número de errores matemáticos. Por el contrario, el primer método utiliza una técnica que está más basada en la fuerza bruta.

Observe que, siempre que tengamos un algoritmo de tipo divide y vencerás que resuelva dos problemas de tamaño mitad con un trabajo lineal adicional, siempre obtendremos un tiempo de ejecución $O(N \log N)$.

7.5.3 Una cota superior general para el tiempo de ejecución de los algoritmos divide y vencerás

La fórmula general dada en esta sección permite que el número de subproblemas, el tamaño de los subproblemas y la cantidad de trabajo adicional estén representados por fórmulas generales. El resultado puede utilizarse sin necesidad de comprender la demostración.

El análisis de la Sección 7.5.2 muestra que, cuando se divide un problema en dos mitades iguales que se resuelve de forma recursiva con un coste adicional $O(N)$, el resultado es un algoritmo $O(N \log N)$. ¿Qué pasa si dividimos un problema en tres problemas de tamaño mitad con un coste adicional lineal, o en siete problemas de tamaño mitad con un coste cuadrático adicional? (*véase* el Ejercicio 7.18). En esta sección vamos a proporcionar una fórmula general para calcular el tiempo de ejecución de un algoritmo de tipo divide y vencerás. La fórmula requiere tres parámetros:

- A, que es el número de subproblemas.
- B, que es el tamaño relativo de los subproblemas (por ejemplo, $B = 2$ representa subproblemas de tamaño mitad).
- k, que es representativo del hecho de que el coste adicional es $\Theta(N^k)$.

La fórmula y su demostración se presentan en el Teorema 7.5. La demostración de la fórmula requiere estar familiarizado con las sumas geométricas. Sin embargo, el conocimiento de la demostración no es necesario para poder utilizar la fórmula.

Teorema 7.5 La solución a la ecuación $T(N) = AT(N/B) + O(N^k)$, donde $A \geq 1$ y $B > 1$, es

$$T(N) = \begin{cases} O(N^{\log_B A}) & \text{para } A > B^k \\ O(N^k \log N) & \text{para } A = B^k \\ O(N^k) & \text{para } A < B^k \end{cases}$$

Antes de demostrar el Teorema 7.5, vamos a examinar algunas aplicaciones. Para el problema de suma máxima de subsecuencia contigua, tenemos dos problemas, dos mitades y un coste adicional de tipo lineal. Los valores aplicables son $A = 2$, $B = 2$ y $k = 1$. Por tanto, se aplica el segundo caso del Teorema 7.5 y obtenemos $O(N \log N)$, lo que concuerda con nuestro cálculo anterior. Si resolvemos de forma recursiva tres problemas de tamaño mitad con un coste lineal adicional, tendremos $A = 3$, $B = 2$ y $k = 1$, y se aplicará el primer caso. El resultado será $O(N^{\log_2 3}) = O(N^{1,59})$. Aquí, el coste adicional no contribuye al coste total del algoritmo. Cualquier coste adicional inferior a $O(N^{1,59})$ nos daría el mismo tiempo de ejecución para el algoritmo recursivo. Un algoritmo que resolviera tres problemas de tamaño mitad, pero requiriera un coste adicional cuadrático, tendría un tiempo de ejecución $O(N^2)$ porque se aplicaría el tercer caso. De hecho, el coste adicional domina en cuanto excede del umbral $O(N^{1,59})$. En el umbral, el coste adicional es el factor logarítmico mostrado en el segundo caso. Ahora, procedamos a demostrar el Teorema 7.5.

Demostración del Teorema 7.5

Siguiendo el método esbozado en la segunda demostración del Teorema 7.4, vamos a suponer que N es una potencia de B y que $N = B^M$. Entonces $N/B = B^{M-1}$ y $N^k = (B^M)^k = (B^k)^M$. Suponemos que $T(1) = 1$ e ignoramos el factor constante $O(N^k)$. Entonces tendremos la siguiente ecuación básica

$$T(B^M) = AT(B^{M-1}) + (B^k)^M$$

Si dividimos entre A^M obtenemos la nueva ecuación básica

$$\frac{T(B^M)}{A^M} = \frac{T(B^{M-1})}{A^{M-1}} + \left(\frac{B^k}{A}\right)^M$$

Ahora podemos escribir esta ecuación para todo M, obteniendo

$$\begin{aligned}
\frac{T(B^M)}{A^M} &= \frac{T(B^{M-1})}{A^{M-1}} + \left(\frac{B^k}{A}\right)^M \\
\frac{T(B^{M-1})}{A^{M-1}} &= \frac{T(B^{M-2})}{A^{M-2}} + \left(\frac{B^k}{A}\right)^{M-1} \\
\frac{T(B^{M-2})}{A^{M-2}} &= \frac{T(B^{M-3})}{A^{M-3}} + \left(\frac{B^k}{A}\right)^{M-2} \qquad \textbf{(7.9)} \\
&\cdots \\
\frac{T(B^1)}{A^1} &= \frac{T(B^0)}{A^0} + \left(\frac{B^k}{A}\right)^1
\end{aligned}$$

Si sumamos el conjunto de ecuaciones representado por la Ecuación 7.9, de nuevo casi todos los términos del lado izquierdo se cancelan con los primeros términos del lado derecho, lo que nos da

$$\begin{aligned}
\frac{T(B^M)}{A^M} &= 1 + \sum_{i=1}^{M} \left(\frac{B^k}{A}\right)^i \\
&= \sum_{i=0}^{M} \left(\frac{B^k}{A}\right)^i
\end{aligned}$$

Por tanto

$$T(N) = T(B^M) = A^M \sum_{i=0}^{M} \left(\frac{B^k}{A}\right)^i \qquad \textbf{(7.10)}$$

Si $A > B^k$, entonces la suma es una serie geométrica con una razón inferior a 1. Puesto que la suma de una serie infinita converge a una constante, esta suma finita también está acotada por una constante. Por tanto, obtenemos

$$T(N) = O(A^M) = O(N^{\log_B A}) \qquad \textbf{(7.11)}$$

Continúa

Demostración del Teorema 7.5 (cont.)

Si $A = B^k$, entonces cada término de la suma de la Ecuación 7.10 es 1. Puesto que la suma contiene $1 + \log_B N$ términos y $A = B^k$ implica que $A^M = N^k$

$$T(N) = O(A^M \log_B N) = O(N^k \log_B N) = O(N^k \log N)$$

Por último, si $A < B^k$, entonces los términos de la serie geométrica con mayores que 1. Podemos calcular la suma utilizando una fórmula estándar, obteniendo

$$T(N) = A^M \frac{\left(\frac{B^k}{A}\right)^{M+1} - 1}{\frac{B^k}{A} - 1} = O\left(A^M \left(\frac{B^k}{A}\right)^M\right) = O((B^k)^M) = O(N^k)$$

lo que demuestra el último caso del Teorema 7.5.

7.6 Programación dinámica

La programación dinámica resuelve subproblemas de manera no recursiva anotando las respuestas en una tabla.

Un problema que pueda ser expresado matemáticamente de forma recursiva también puede ser expresado como un algoritmo recursivo. En muchos casos, el hacer esto nos proporciona una significativa mejora en el rendimiento, por comparación con las búsquedas exhaustivas más simples. Cualquier fórmula matemática recursiva podría traducirse directamente a un algoritmo recursivo, pero a menudo el compilador no hará justicia al algoritmo recursivo y el resultado será un programa ineficiente. Este es el caso por ejemplo del cálculo recursivo de los números de Fibonacci que hemos descrito en la Sección 7.3.4. Para evitar esta explosión recursiva, podemos utilizar la *programación dinámica* para reescribir el algoritmo recursivo en forma de un algoritmo no recursivo, que vaya anotando sistemáticamente en una tabla las respuestas a los subproblemas. Ilustraremos esta técnica con el siguiente problema.

Problema del cambio de moneda

Para una moneda nacional en la que haya monedas físicas de C_1, C_2, ..., C_N (céntimos), ¿cuál es el número mínimo de monedas necesarias para obtener K céntimos de cambio?

Los algoritmos voraces toman decisiones localmente óptimas en cada paso. Esta es la cosa más sencilla que podemos hacer, pero no siempre es la correcta.

La moneda nacional de Estados Unidos tiene monedas de 1, 5, 10 y 25 centavos (ignore la moneda de 50 centavos que se usa de forma menos frecuente). Podemos obtener 63 centavos utilizando dos monedas de 25 centavos, una moneda de 10 centavos y tres monedas de 1 centavo, lo que nos da un total de seis monedas. El obtener el cambio necesario con este conjunto de monedas es relativamente sencillo: basta con utilizar repetidamente la moneda de mayor valor que tengamos disponible. Se puede demostrar que para el caso de los Estados Unidos, esta técnica siempre minimiza el número total de monedas utilizadas, lo que es un ejemplo de lo que se denomina algoritmos voraces. En un *algoritmo voraz*, se toma durante cada fase una decisión que parece ser óptima sin ocuparse de las consecuencias futuras. Esta estrategia de "aprovechar ahora lo que puedas" es lo que da el nombre a