CAPÍTULO

7 Recursión

Un método parcialmente definido en términos de sí mismo recibe el nombre de *recursivo*. Como muchos otros lenguajes, Java soporta métodos recursivos. La recursión, consistente en el uso de métodos recursivos, es una herramienta de programación potente que en muchos casos puede producir algoritmos cortos y eficientes. Este capítulo estudia cómo funciona la recursión, proporcionando una visión de sus variaciones, sus limitaciones y algunos de sus numerosos usos. La discusión sobre la recursión comienza con un examen del principio matemático en el que se basa: la *inducción matemática*. Después se proporcionan ejemplos de métodos recursivos simples y se demuestra que éstos generan respuestas correctas.

En este capítulo veremos:

- Las cuatro reglas básicas de la recursión, y varios ejemplos sencillos.
- Aplicaciones numéricas de la recursión que nos conducen a la implementación de un algoritmo de encriptación.
- Una técnica general llamada *divide y vencerás*.
- Una técnica general llamada *programación dinámica* que es similar a la recursión pero que usa tablas en lugar de llamadas recursivas.
- Una técnica general llamada *vuelta atrás* que consiste en una búsqueda sistemática y exhaustiva.

7.1 ¿Qué es la recursión?

Un *método recursivo* es un método que, directa o indirectamente, se hace una llamada a sí mismo. Esto puede parecer un círculo vicioso: ¿cómo un método F puede resolver un problema llamándose a sí mismo? La clave está en que el método F se llama a sí mismo pero con instancias diferentes, más simples, en algún sentido adecuado. Aquí tiene algunos ejemplos:

Un *método recursivo* es un método que, directa o indirectamente, se hace una llamada a sí mismo.

- Los ficheros en un computador se almacenan generalmente en directorios. Los usuarios pueden crear directorios, que a su vez almacenen más ficheros y directorios. Suponga que deseamos examinar cada fichero de un directorio D, incluyendo todos los ficheros de sus subdirectorios (y sus subdirectorios, y así sucesivamente). Esto se puede hacer examinando recursivamente los ficheros de cada subdirectorio junto con todos los ficheros en el directorio D (este problema se estudia en el Capítulo 17).
- Supongamos que tenemos un diccionario grande. Las palabras en los diccionarios se definen en términos de otras palabras. Cuando buscamos el signifi-

cado de una palabra, no siempre entendemos la definición, por lo que debemos buscar algunas palabras de la propia definición. De la misma forma, puede que no entendamos algunas de ellas, en cuyo caso tendremos que continuar la búsqueda durante un rato. Como el diccionario es finito, eventualmente llegaremos a un punto en el que entendemos todas las palabras de una definición, con lo que entendemos dicha definición y podemos volver hacia atrás a través de las otras definiciones, o bien nos daremos cuenta de que las definiciones son circulares por lo que nos quedamos atascados, o bien alguna palabra que no entendemos no está definida en el diccionario. Nuestra estrategia recursiva para comprender las palabras es la siguiente: si conocemos el significado de una palabra, ya hemos terminado; en caso contrario, buscamos la palabra en el diccionario. Si entendemos todas las palabras de la definición, ya hemos terminado; en caso contrario, averiguamos el significado de la definición buscando recursivamente las palabras que no conocemos. Este procedimiento terminará si el diccionario está bien definido, pero puede ciclar indefinidamente si una palabra está definida circularmente.

- Habitualmente (partes de) los lenguajes de programación se definen de forma recursiva. Por ejemplo, una expresión (aritmética) es una variable primitiva, o una expresión con paréntesis, o la suma de dos expresiones, etc.

La recursión es una herramienta muy potente de resolución de problemas. Muchos algoritmos se pueden expresar más fácilmente usando una formulación recursiva. Y lo que es más, hay muchos problemas cuyas soluciones más eficientes usan esta formulación recursiva natural. Pero debemos ser cuidadosos para no crear una lógica circular que produzca bucles infinitos.

En este capítulo se estudian las condiciones generales que deben satisfacer los algoritmos recursivos, ofreciéndose varios ejemplos prácticos. También muestra que en ocasiones los algoritmos que se expresan de una forma natural en modo recursivo deben rescribirse sin recursión para conseguir mayor eficiencia.

7.2 Fundamentos: demostraciones por inducción matemática

La *inducción* es una técnica importante de demostración que se usa para demostrar teoremas que cumplen los enteros positivos.

En esta sección se estudia la técnica de demostración mediante *inducción matemática*. (A lo largo de este capítulo, se omitirá la palabra *matemática* al referirnos a esta técnica.) Las demostraciones por inducción se usan habitualmente para demostrar teoremas que cumplen los enteros positivos. Para ilustrar la técnica, comenzaremos por demostrar un teorema sencillo. Sería fácil demostrar este teorema usando otros métodos, pero sucede bastante a menudo que una prueba por inducción es el mecanismo más sencillo.

Teorema 7.1 *Para todo entero $N \geq 1$, la suma de los N primeros enteros, dada por $\sum_{i=1}^{N} i = 1 + 2 + \ldots + N$, es igual a $N(N+1)/2$.*

Es fácil ver que el teorema es cierto para $N = 1$ porque tanto el lado izquierdo como el lado derecho se evalúan a 1. Un estudio más laborioso revelaría, por ejemplo, que es cierto para $2 \leqslant N \leqslant 10$. Sin embargo, el hecho de que el teorema se cumpla para todos los N para los cuales es fácil comprobarlo *a mano*, no implica que sea cierto para todo N. Consideremos, por ejemplo, los números de la forma $2^{2^k} + 1$. Los primeros cinco números (correspondientes a $0 \leqslant k \leqslant 4$) son 3, 5, 17, 257 y 65.537, los cuales son todos primos. De hecho, hubo un momento en el que se conjeturó que todos los números de esta forma debían ser primos. Pero ello no es cierto: se puede comprobar fácilmente con un computador que $2^{2^5} + 1 = 641 \times 6.700.417$. De hecho, fuera de los citados, no se conoce ningún otro primo de la forma $2^{2^k} + 1$.

Una prueba por inducción trabaja en dos pasos. Primero, al igual que antes, se demuestra que el teorema es cierto para los casos más sencillos. Después, se demuestra que si el teorema es cierto para ciertos casos, entonces se puede extender para incluir el siguiente caso. Por ejemplo, se demuestra que un teorema que es cierto para todo $1 \leqslant N \leqslant k$ también debe ser cierto para $1 \leqslant N \leqslant k + 1$. Una vez se muestra cómo extender el rango de casos ciertos, queda demostrado el resultado para todos los casos, ya que podemos extender el rango de los casos ciertos indefinidamente. Veamos la utilización de esta técnica para demostrar el Teorema 7.1.

Una *demostración por inducción* muestra que el teorema es cierto para algunos casos simples y después muestra cómo extender la certeza indefinidamente.

Demostración (del Teorema 7.1)

Claramente, el teorema es cierto para $N = 1$. Supongamos que el teorema es cierto para todo $1 \leqslant N \leqslant k$. Entonces

$$\Sigma_{i=1}^{k+1} i = (k + 1) + \Sigma_{i=1}^{k} i. \qquad \textbf{(7.1)}$$

Puesto que por hipótesis el teorema es cierto para k, podemos sustituir la suma del lado derecho de la Ecuación 7.1 por $k\,(k+1)/2$, obteniendo

$$\Sigma_{i=1}^{k+1} i = (k + 1) + k(k + 1)/2. \qquad \textbf{(7.2)}$$

Una manipulación algebraica del lado derecho de la Ecuación 7.2 produce ahora

$$\Sigma_{i=1}^{k+1} i = (k + 1)\,(k + 2)/2.$$

Esto confirma el teorema para el caso $k + 1$. Luego, por inducción, el teorema es cierto para todos los enteros $N \geqslant 1$.

En una demostración por inducción, se llama *caso base* al caso sencillo que se puede demostrar directamente.

La *hipótesis de inducción* supone que el teorema es cierto para un caso arbitrario.

¿Por qué constituye esto una prueba? En primer lugar, el teorema es cierto para $N = 1$, al cual se le llama *caso base*. Se puede ver como la base de nuestra confianza en que el teorema es cierto en general. El caso base es el caso (o casos) sencillo que demostramos directamente. Una vez establecido el caso base, podemos suponer que el teorema es cierto para un k arbitrario, suposición que recibe el nombre de hipótesis de inducción. Entonces se demuestra que si el teorema es cierto para k, también es cierto para $k + 1$. En nuestro caso, aplicando el proceso, puesto que sabemos que el teorema es cierto para el caso base $N = 1$, obtenemos que es cierto para $N = 2$. Puesto que es cierto para $N = 2$, también debe serlo para

demostrándose que, a partir de esta hipótesis, podemos probar que el teorema es cierto para el siguiente caso. Esto representa el paso inductivo de la demostración.

$N = 3$, y por tanto para $N = 4$, y así sucesivamente. De esta forma vemos que el teorema es cierto para todo entero positivo a partir de $N = 1$.

Apliquemos la inducción a un segundo problema no tan simple como el anterior. Consideramos la secuencia de números 1^2, $2^2 - 1^2$, $3^2 - 2^2 + 1^2$, $4^2 - 3^2 + 2^2 - 1^2$, $5^2 - 4^2 + 3^2 - 2^2 + 1^2$, etc. Cada miembro representa la suma con alternancia de signos de los N primeros cuadrados. La secuencia se evalúa a 1, 3, 6, 10 y 15. Parece que, en general, la suma será exactamente igual a la de los N primeros enteros, que, por el Teorema 7.1, es $N(N+1)/2$. El Teorema 7.2 demuestra este hecho.

Teorema 7.2

La suma $\Sigma_{i=N}^{1}(-1)^{N-i}i^2 = N^2 - (N-1)^2 + (N-2)^2 \ldots$ *es igual a* $N(N+1)/2$.

Demostración

La demostración es por inducción.

Caso base: claramente, el teorema es cierto para $N = 1$.

Hipótesis de inducción: supondremos que el teorema es cierto para k: $\Sigma_{i=k}^{1}(-1)^{k-i}i^2 = \dfrac{k(k+1)}{2}$.

Paso inductivo: debemos demostrar que el resultado es cierto para $k+1$; *es decir, que* $\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = \dfrac{(k+1)(k+2)}{2}$. *Escribimos*

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - k^2 + (k-1)^2 \ldots . \qquad \textbf{(7.3)}$$

Si rescribimos el lado derecho de la Ecuación 7.3 obtenemos

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - (k^2 - (k-1)^2 \ldots).$$

Esto permite una sustitución, obteniéndose

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - (\Sigma_{i=k}^{1}(-1)^{k-i}i^2) \qquad \textbf{(7.4)}$$

Aplicando la hipótesis de inducción, podemos sustituir el sumatorio del lado derecho de la Ecuación 7.4, obteniendo

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - k(k+1)/2. \qquad \textbf{(7.5)}$$

Una manipulación algebraica simple del lado derecho de la Ecuación 7.5 produce ahora

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)(k+2)/2,$$

lo cual demuestra el teorema para $N = k+1$. *Luego, por inducción, el teorema es cierto para todo* $N \geqslant 1$.

7.3 Recursión básica

Las demostraciones por inducción nos muestran que si sabemos que una afirmación es cierta para los casos más sencillos, y podemos demostrar que la certeza para un caso implica la del siguiente, entonces sabemos que la afirmación es cierta para todos los casos.

En ocasiones las funciones matemáticas se definen recursivamente. Por ejemplo, si $S(N)$ denota la suma de los N primeros enteros, tenemos $S(1) = 1$, y podemos escribir $S(N) = S(N - 1) + N$. Hemos así definido la función S en términos de una instancia más pequeña de sí misma. La definición recursiva de $S(N)$ es equivalente a la forma explícita $S(N) = N(N + 1)/2$, excepto por el hecho de que la definición recursiva solamente vale para enteros positivos y su valor no puede calcularse de forma tan directa.

Un método recursivo simple se define en términos de una instancia más pequeña de sí mismo. Debe haber algún caso base que se pueda resolver sin nuevas llamadas recursivas.

Algunas veces es más fácil definir una función recursivamente que hacerlo en forma explícita. En la Figura 7.1 se muestra una implementación directa de la función recursiva S. El caso $N = 1$ es el caso base, para el que sabemos que $S(1) = 1$. Este caso se trata en las líneas 5 y 6. Es el caso base de la recursión, pues es la instancia que se puede resolver sin llamadas recursivas. En otro caso, aplicamos la definición recursiva $S(N) = S(N - 1) + N$, situada en la línea 8. Es difícil imaginar una forma más sencilla de implementar la función recursiva. Entonces, la pregunta natural es, ¿funciona esto realmente?

La respuesta es sí. Con las limitaciones que discutiremos en unos momentos, esta rutina funciona correctamente. Veamos cómo se evalúa la llamada a `s(4)`. Cuando se llama a `s(4)`, el test de la línea 5 falla, por lo que se ejecuta la línea 8, donde evaluamos `s(3)`. Como en el caso de otro método cualquiera, esto requiere una llamada a `s`, con lo que llegamos a la línea 5, donde el test falla, por lo que vamos a la línea 8. En este punto llamamos a `s(2)`, con lo que, de nuevo, llamamos a `s` siendo ahora `n` igual a 2. El test de la línea 5 vuelve a fallar, por lo que llamamos a `s(1)` en la línea 8. Ahora tenemos que `n` es igual a 1, con lo que `s(1)` devuelve 1. En este punto, puede continuar la evaluación de `s(2)`, sumando el valor de `s(1)` a 2, con lo que `s(2)` devuelve 3. Ahora continúa la evaluación de `s(3)` sumando el valor devuelto por `s(2)` a `n`, que vale 3, con lo que `s(3)` devuelve 6. Esto permite completar la llamada a `s(4)`, que devuelve 10 finalmente.

Observe que aunque parece que `s` se está llamando a sí misma, podemos interpretar que en realidad está llamando a un *clon* de sí misma. Dicho clon se puede ver como un método distinto que usa un parámetro diferente. En cada momento solamente hay un clon activo; el resto están suspendidos. Es trabajo del computador y no de uno llevar a cabo las tareas de bajo nivel. Es cierto que si hubiera

```
1  // Evalúa la suma de los n primeros enteros
2
3  public static long s( int n )
4  {
5      if( n == 1 )
6          return 1;
7      else
8          return s( n - 1 ) + n;
9  }
```

Figura 7.1 Evaluación recursiva de la suma de los N primeros enteros.

demasiadas, incluso para un computador, sería el momento de preocuparse. Esas cuestiones se discutirán más adelante en este mismo capítulo.

El *caso base* es una instancia que se puede resolver sin recursión. Cualquier llamada recursiva debe progresar hacia un caso base.

Si tenemos un caso base y nuestras llamadas recursivas progresan hacia el caso base, entonces terminaremos eventualmente. Con ello concluimos nuestras dos primeras reglas fundamentales de la recursión:

1. *Casos base:* se debe tener siempre al menos un caso base que pueda resolverse sin recursión.
2. *Progreso:* cualquier llamada recursiva debe progresar hacia un caso base.

Nuestra rutina tiene algunos problemas. Un primer problema lo constituiría la llamada a `s(0)`, para la cual el método se comporta de forma pobre[1]. Sin embargo, no debe verse como un error porque el caso recursivo de la definición de $S(N)$ no incluye los valores $N < 1$. Podríamos arreglar este problema extendiendo la definición de $S(N)$ para incluir $N = 0$. Puesto que en este caso no hay números que sumar, un valor natural para $S(0)$ sería 0. Esto además haría que la definición recursiva se pudiera aplicar a $S(1)$, ya que $S(0) + 1$ es 1. Para implementar este cambio, basta sustituir `1` por `0` en las líneas 5 y 6. Los N negativos también producirían comportamientos indeseables, que se podrían solventar de forma análoga a como hicimos para $N = 0$ (se deja como Ejercicio 7.2).

Un segundo problema es que cuando el parámetro `n` es grande, el programa puede abortar o no acabar. En nuestro sistema, por ejemplo, no se puede tratar $N \geqslant 9.410$.

Esto sucede porque, como hemos visto, la implementación de la recursión requiere un tratamiento de las llamadas recursivas pendientes, y para cadenas de recursión lo suficientemente largas, el computador simplemente se queda sin memoria dedicada al efecto. Esto se explicará con más detalle más adelante en el capítulo. Es también cierto que la rutina es más costosa que el bucle equivalente, pues el tratamiento de las llamadas recursivas lleva un cierto tiempo.

Queda claro pues que este ejemplo concreto no representa el mejor uso de la recursión, puesto que es muy sencillo resolver el problema sin ella. La mayor parte de los buenos usos de la recursión no agotarán la memoria del computador y sólo son ligeramente más lentos que las implementaciones no recursivas. Pero la recursión siempre nos conduce a producir un código más compacto.

7.3.1 Impresión de números en cualquier base

Un buen ejemplo de cómo la recursión simplifica la codificación de los procedimientos es la impresión de números. Supongamos que queremos imprimir un número N no negativo en base diez pero que no tenemos disponible un método de impresión de números, aunque sí podemos imprimir un dígito cada vez. Consideremos, por ejemplo, cómo imprimiríamos el número `1369`. Primero necesitaríamos imprimir un `1`, luego un `3`, después un `6` y finalmente un `9`. El problema es que obtener el primer dígito es un tanto costoso: dado un número `n`, necesitamos un bucle para determinar el primer dígito de `n`. Esto contrasta con la obtención del

[1] Se realiza una llamada a `s(-1)`, y el programa falla eventualmente por haber demasiadas llamadas recursivas pendientes. Tenemos el problema de que no hay progreso hacia un caso base.

último dígito, que se puede calcular de forma inmediata como `n % 10` (que vale `n` para `n` menor que 10).

La recursión nos proporciona una solución elegante. Para imprimir `1369`, bastaría con imprimir `136` seguido del último dígito, `9`. Como hemos mencionado, es fácil obtener el último dígito usando el operador `%`. Imprimir el número entero exceptuando su último dígito es también sencillo, ya que corresponde a imprimir `n/10`, lo que se puede hacer mediante una llamada recursiva.

El método de la Figura 7.2 implementa esta rutina de impresión. Si `n` es menor que 10, entonces no se ejecuta la línea 6 y solamente se imprime el dígito `n%10`. En caso contrario, se imprime todos menos el último dígito y después se imprime éste.

Observe que tenemos un caso base (`n` es un entero de un solo dígito) y que todas las llamadas recursivas progresan hacia el caso base porque el argumento con el que se hace la llamada recursiva tiene un dígito menos. Por tanto, hemos satisfecho las dos primeras reglas fundamentales de la recursión.

Para que nuestra rutina de impresión sea útil más allá de la mera anécdota, la extendemos para que pueda imprimir el número en cualquier base entre 2 y 16[2]. Esta modificación se muestra en la Figura 7.3. Hemos introducido un vector de caracteres para hacer más sencilla la impresión de las cifras entre la `a` y la `f`, por lo que ahora la salida de cada dígito se realiza indexando el vector `tablaDigitos`. Obsérvese que la rutina `imprimeEntero` no es robusta, pues si `base` es mayor que 16, entonces el índice dentro de `tablaDigitos` estará fuera de rango. Por su parte, si `base` es 0, se producirá un error aritmético en la línea 9 al intentar realizar una división por 0.

```
1  // Imprime n como un número en base diez
2
3  public static void imprimeBaseDiez( int n )
4  {
5      if( n >= 10 )
6          imprimeBaseDiez( n / 10 );
7      System.out.print( '0' + n % 10 );
8  }
```

Figura 7.2 Rutina recursiva para imprimir *N* en base diez.

```
1  // Imprime n en cualquier base
2  // Asume que 2 <= base <= 16
3
4  final static String tablaDigitos = "0123456789abcdef";
5
6  public static void imprimeEntero( int n, int base )
7  {
8      if( n >= base )
9          imprimeEntero( n / base, base );
10     System.out.print( tablaDigitos.charAt( n % base ) );
11 }
```

Figura 7.3 Rutina recursiva para imprimir *N* en cualquier base entre 2 y 16.

[2] El método `toString` de Java puede tomar cualquier base, pero muchos lenguajes no tienen esta capacidad predefinida.

Un fallo en el progreso hacia el caso base hace que el programa no funcione.

Una *rutina guía* comprueba la validez de la llamada original y genera la llamada a la rutina recursiva.

La situación errónea más interesante aparece cuando `base` es 1. Cuando tal cosa sucede, la llamada recursiva de la línea 9 no progresa hacia el caso base, pues los dos parámetros de la llamada recursiva son idénticos a los de la llamada original. Por ello el sistema hará repetidas llamadas recursivas hasta que se quede sin espacio disponible, momento en el que terminará de forma poco elegante lanzando una excepción.

Podemos conseguir que la rutina sea más robusta añadiendo una comprobación explícita del valor de `base`. El problema con dicha estrategia es que la comprobación se realizaría en cada una de las llamadas recursivas a `imprimeEntero`, y no solamente en la primera llamada. Una vez que `base` es válida en la primera llamada, es una tontería volver a comprobarlo, pues su valor no cambia a lo largo de las llamadas recursivas, por lo que seguirá siendo válida. Una forma de evitar esta ineficiencia consiste en introducir una rutina guía. Dicha *rutina guía* comprueba la validez de `base` y tras ello llama a la rutina recursiva. En la Figura 7.4 se presenta el programa de impresión completo. El uso de rutinas guía sobre programas recursivos es una técnica habitual.

```
1       // Imprime n en cualquier base
2
3   static final String tablaDigitos = "0123456789abcdef";
4   private static final int maxBase = tablaDigitos.length( );
5
6       // Rutina recursiva
7   public static void imprimeEnteroRec( int n, int base )
8   {
9       if( n >= base )
10          imprimeEnteroRec( n / base, base );
11      System.out.print( tablaDigitos.charAt( n % base ) );
12  }
13
14      // Rutina guía
15  public static void imprimeEntero( int n, int base )
16  {
17      if( base <= 1 || base > maxBase )
18          System.err.println( "Base ilegal: " + base );
19      else
20      {
21          if( n < 0 )
22          {
23              n = -n;
24              System.out.print( "-" );
25          }
26          imprimeEnteroRec( n, base );
27      }
28  }
```

Figura 7.4 Programa robusto de impresión de números.

7.3.2 ¿Por qué funciona?

En esta sección se demuestra, de una forma rigurosa, que el algoritmo `imprimeBaseDiez` funciona correctamente. Nuestro objetivo es verificar que el algoritmo es correcto, por lo que asumiremos que no hemos cometido ningún error en su codificación.

Teorema 7.3

El algoritmo `imprimeBaseDiez` *de la Figura 7.2 imprime correctamente* `n` *en base 10.*

Demostración

Sea k el número de dígitos en base 10 de `n`*. La demostración se realiza por inducción sobre k.*

Caso base: si $k = 1$*, no se realiza ninguna llamada recursiva, y la línea 7 produce correctamente como salida el único dígito de* `n`*.*

Hipótesis de inducción: supongamos que `imprimeBaseDiez` *funciona correctamente para todos los enteros de* $k \geqslant 1$ *dígitos.*
Paso inductivo: veamos que a partir de estas hipótesis podemos probar la corrección del algoritmo para cualquier entero `n` *de* $k + 1$ *dígitos. Puesto que* $k \geqslant 1$*, la condición de la instrucción* `if` *de la línea 5 se satisface para todo entero* `n` *de* $k + 1$ *dígitos.*
Por hipótesis de inducción, la llamada recursiva de la línea 6 imprime los k primeros dígitos de `n`*. Entonces la llamada de la línea 7 imprime el último dígito. Por tanto si cualquier entero de k dígitos se imprime correctamente, también se imprimirá correctamente uno de* $k + 1$ *dígitos. Concluimos entonces que* `imprimeBaseDiez` *funciona para todo k, y por tanto para todo* `n`*.*

Se puede demostrar la corrección de los algoritmos recursivos usando inducción matemática.

La demostración del Teorema 7.3 ilustra un principio importante. Cuando se diseña un algoritmo recursivo, siempre podemos asumir que las llamadas recursivas internas funcionan correctamente, y dicha suposición podrá utilizarse como hipótesis de inducción al hacer la demostración de la corrección.

A primera vista, tal asunción puede parecer extraña. Pero recuerde que al razonar estructuradamente sobre un programa complejo no recursivo, siempre hemos podido suponer que las llamadas a los métodos funcionan correctamente, y por tanto la suposición de que las llamadas recursivas funcionan no es muy diferente. Como cualquier método, una rutina recursiva necesita combinar las soluciones producidas por las llamadas a otros métodos para producir una solución, con la particularidad de que tales métodos pueden ser instancias más sencillas del método original.

Esta observación nos conduce a la tercera regla fundamental de la recursión:

3. *«Puede creerlo»:* asuma siempre que toda llamada recursiva interna funciona correctamente.

La tercera regla fundamental de la recursión es: asuma siempre que toda llamada recursiva interna funciona correctamente. Use esta regla para diseñar sus algoritmos recursivos.

La Regla 3 nos dice que cuando diseñamos un método recursivo, no tenemos que intentar seguir el, posiblemente largo y tortuoso, camino de llamadas recursivas. Como vimos anteriormente, ésta puede ser una tarea tediosa que haría muy difícil el diseño y verificación. De hecho, en muchas ocasiones, aunque hagamos un buen uso de la recursión, dicha traza puede ser casi imposible de comprender. Intuitivamente, estamos dejando que la máquina se ocupe de las tareas de bajo nivel (las llamadas recursivas internas), ya que si nos ocupáramos nosotros directamente, el código resultante sería muchísimo más largo.

Este principio es tan importante que no nos cansaremos de repetirlo: *asuma siempre que toda llamada recursiva interna funciona correctamente.*

7.3.3 Cómo funciona

Las tareas de bajo nivel en un lenguaje procedimental u orientado a objetos se llevan a cabo a través de una *pila de registros de activación*. La recursión es una consecuencia natural.

Recuerde que la implementación de la recursión requiere la realización de tareas de bajo nivel por parte del computador. Dicho de otra forma, la implementación de cualquier método requiere ciertas tareas de bajo nivel y un método recursivo no es en ello particularmente especial (excepto por el hecho de que puede sobrepasar las limitaciones del computador al llamarse a sí misma demasiadas veces).

Al igual que otros lenguajes como C++ y Ada, Java implementa los métodos mediante una *pila de registros de activación*. Podemos ver un registro de activación como un trozo de papel que contiene información relevante sobre el método, la cual incluye, por ejemplo, los valores de los parámetros y las variables locales. El contenido concreto de los registros de activación depende del sistema.

Las secuencias de llamada a un método y retorno de un método son operaciones sobre una pila.

Se usa una pila de registros de activación porque la terminación de los métodos se produce en sentido inverso a la invocación de los mismos. Recuerde que las pilas son adecuadas para invertir secuencias. Lo más habitual es que la cima de la pila almacene el registro de activación del método actualmente activo. Cuando se llama al método *G*, se coloca un nuevo registro de activación en la cima de la pila, lo cual hace que *G* sea el método actualmente activo. Cuando termina un método, se desapila la cima de la pila y el registro de activación que queda en la cima contiene los valores previos.

Como ejemplo, en la Figura 7.5 se muestra una pila de registros de activación que aparecen en el curso de la evaluación de `s(4)`. En este punto, tenemos llamadas pendientes a `main`, `s(4)` y `s(3)`, y estamos procesando `s(2)`.

Se produce una sobrecarga en espacio debido a la memoria necesaria para almacenar un registro de activación para cada método actualmente activo. Así, en el ejemplo anterior en el que `s(9410)` aborta, el sistema tenía espacio reservado para unos 9.410 registros de activación. (Observe que `main` genera también un registro de activación.) La sobrecarga en tiempo producida al ejecutar una llamada a un método corresponde a la ejecución de las operaciones de apilar y desapilar sobre la pila interna.

La recursión siempre se puede eliminar usando una pila. Esto, en raras ocasiones, podría ser necesario para ahorrar espacio.

La cercana relación entre recursión y pilas nos dice que los programas recursivos siempre se pueden implementar de forma iterativa con una pila explícita. Presumiblemente la pila almacenará objetos más pequeños que los registros de activación, dada la generalidad de éstos, por lo que podemos esperar razonablemente que se use algo menos de espacio. Además, el resultado sería ligeramente más rápido, pero el código notablemente más largo y complejo. Los modernos compiladores optimizantes han reducido los costes en tiempo asociados a la recursión a un grado tal que raramente vale la pena eliminar la recursión de una aplicación que la usa correctamente.

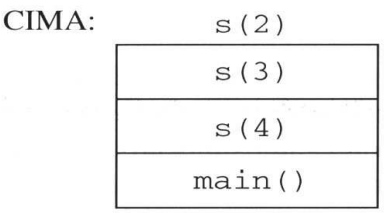

Figura 7.5 Pila de registros de activación.

7.3.4 Demasiada recursión puede ser peligrosa

No use recursión en sustitución de un simple bucle.

En este libro hay muchos ejemplos que ilustran el poder de la recursión. Sin embargo, antes de examinar esos ejemplos, es importante darse cuenta de que la recursión no siempre es apropiada. Por ejemplo, el uso de la recursión en la Figura 7.1 es pobre, un bucle funcionaría igual de bien. El inconveniente práctico es que las llamadas recursivas consumen tiempo y limitan el valor de `n` para el cual se puede ejecutar el programa. Una buena regla es que nunca se debe usar la recursión en sustitución de un simple bucle.

El *i*-ésimo *número de Fibonacci* es la suma de los dos anteriores números de Fibonacci.

Un problema mucho más serio aparece, por ejemplo, al intentar calcular los números de Fibonacci recursivamente. Los primeros *números de Fibonacci* F_0, F_1, ..., F_i se definen de la siguiente forma: $F_0 = 0$ y $F_1 = 1$. El *i*-ésimo número de Fibonacci es igual a la suma de los dos anteriores, es decir $F_i = F_{i-1} + F_{i-2}$. A partir de esta definición, podemos determinar que la serie de números de Fibonacci continúa así: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Los números de Fibonacci tienen una enormidad de propiedades, que en muchos casos parecen surgir de forma inesperada. De hecho, la revista *The Fibonacci Quarterly*[3] existe solamente con el propósito de publicar teoremas que impliquen a los números de Fibonacci. Por ejemplo, la suma de los N primeros números de Fibonacci es uno menos que F_{N+2}, y la suma de los cuadrados de dos números de Fibonacci consecutivos es siempre otro número de Fibonacci (en el Ejercicio 7.9 se muestran otras igualdades interesantes).

No repita trabajo de forma recursiva: su programa será increíblemente ineficiente.

Puesto que los números de Fibonacci se definen recursivamente, parece natural escribir una rutina recursiva para determinar F_N. Dicha rutina, mostrada en la Figura 7.6, funciona correctamente a nivel teórico, pero tiene un serio problema: en una máquina relativamente rápida tardamos varios minutos en calcular F_{40}. Esto es una cantidad absurda de tiempo considerando que el cálculo requerido precisa solamente de 39 sumas.

El problema subyacente es que esta rutina recursiva lleva a cabo multitud de cálculos repetidos. Para calcular `fib(n)`, calculamos recursivamente `fib(n-1)`. Cuando la llamada recursiva termina, calculamos `fib(n-2)` usando otra llamada recursiva. Pero nosotros ya habíamos calculado `fib(n-2)` al calcular `fib(n-1)`, luego la llamada a `fib(n-2)` es un cálculo repetido y, por ello, una pérdida de tiempo. De hecho se están haciendo dos llamadas a `fib(n-2)` en lugar de una.

```
1 // Calcula el n-ésimo número de Fibonacci
2 // Es un mal algoritmo
3 public static long fib( int n )
4 {
5     if( n <= 1 )
6         return n;
7     else
8         return fib( n - 1 ) + fib( n - 2 );
9 }
```

Figura 7.6 Rutina recursiva para el cálculo de los números de Fibonacci: una mala idea.

[3] *N. del T.:* Una publicación científica trimestral de alcance internacional *¿editorial?*.

Normalmente, hacer dos llamadas a un método en lugar de una, doblaría el tiempo de ejecución de un programa, pero en este caso la situación es peor, porque tanto la llamada a `fib(n-1)` como las llamadas a `fib(n-2)` hacen a su vez llamadas a `fib(n-3)`, lo que significa que hay en realidad tres llamadas a `fib(n-3)`. A partir de ahí, la situación se vuelve aún peor: las llamadas a `fib(n-2)` y a `fib(n-3)` producen llamadas a `fib(n-4)`, cinco en total. Obtenemos por tanto un efecto acumulativo: cada llamada recursiva produce más y más trabajo redundante.

La rutina recursiva `fib` tiene un coste exponencial.

Sea $C(N)$ el número de llamadas a `fib` durante la evaluación de `fib(n)`. Claramente, $C(0) = C(1) = 1$. Para $N \geqslant 2$, hacemos la llamada a `fib(n)`, más todas las llamadas que se necesitan para evaluar `fib(n-1)` y `fib(n-2)`, recursiva e independientemente. Luego, $C(N) = C(N-1) + C(N-2) + 1$. Por inducción, podemos fácilmente probar que para $N \geqslant 3$ se tiene $C(N) = F_{N+2} + F_{N-1} - 1$. En consecuencia, el número de llamadas recursivas es todavía mayor que el número de Fibonacci que queremos calcular y por ello es exponencial. Por ejemplo, para $N = 40$, $F_{40} = 102.334.155$, mientras que el número total de llamadas recursivas es mayor que 300 millones. No se sorprenda si su programa no termina. En la Figura 7.7 se ilustra el crecimiento explosivo del número de llamadas recursivas.

La última regla fundamental de la recursión: nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

Este ejemplo ilustra la cuarta y última regla básica de la recursión:

4. *Regla de interés compuesto:* nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

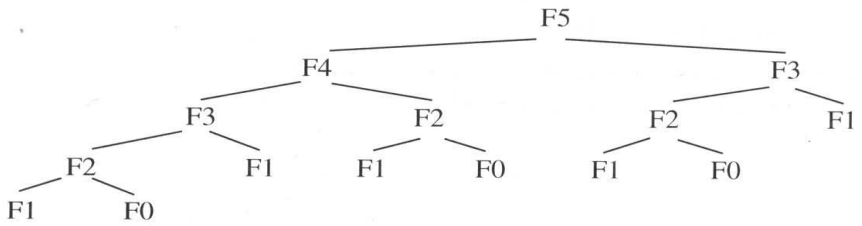

Figura 7.7 Traza del cálculo recursivo de los números de Fibonacci.

7.3.5 Ejemplos adicionales

Quizá la mejor forma de entender la recursión sea viendo algunos ejemplos significativos. En esta sección, presentamos cuatro ejemplos más de recursión. Los dos primeros se podrían implementar fácilmente de forma no recursiva, pero los dos últimos ponen al descubierto el poder de la recursión.

El factorial

Recuerde que $N!$ es el producto de los N primeros enteros, por lo que podemos escribir $N!$ como N veces $(N-1)!$. Combinado con el caso base $1! = 1$, ya tenemos todo lo necesario para una implementación recursiva, la cual se muestra en la Figura 7.8.

```
1  // Evalúa n!
2  public static long factorial( int n )
3  {
4      if( n <= 1 ) // caso base
5          return 1;
6      else
7          return n * factorial( n - 1 );
8  }
```

Figura 7.8 Implementación recursiva de la función `factorial`.

Búsqueda binaria

En la Sección 5.6.2 describimos la búsqueda binaria. Recuerde que en una búsqueda binaria, llevamos a cabo una búsqueda en un vector ordenado A examinando el elemento del centro. Si dicho elemento es el que buscamos, ya hemos terminado. En caso contrario, si el elemento que estamos buscando es menor que el elemento del centro, buscamos en la mitad izquierda del vector, y si es mayor, buscamos en la mitad derecha. Esto, siempre que el vector en cuestión no sea vacío, pues si lo es, entonces concluimos que el elemento buscado no se encuentra en el vector.

Esto se traduce directamente en el método recursivo de la Figura 7.9. El código ilustra la técnica en la que la rutina guía pública hace una llamada inicial a una rutina recursiva privada, y devuelve el resultado recibido (o relanza una excepción). Aquí, la guía inicializa los límites inferior y superior del fragmento del vector a `0` y `a.length-1`.

```
1  /**
2   * Lleva a cabo la búsqueda binaria estándar
3   * usando dos comparaciones por nivel.
4   * Esta es una guía que llama al método recursivo.
5   */
6  public static int busquedaBinaria( Comparable [ ] a,
7                                     Comparable x ) throws ElementoNoEncontrado
8  {
9      return busquedaBinaria( a, x, 0, a.length -1 );
10 }
11
12 /**
13  * Rutina recursiva oculta.
14  */
15 private static int busquedaBinaria( Comparable [ ] a,
16                                     Comparable x, int inf, int sup )
17                                                   throws ElementoNoEncontrado
18 {
19     if( inf > sup )
20         throw new ElementoNoEncontrado( "La búsqueda binaria falla" );
21     int med = ( inf + sup ) / 2;
22     if( a[ med ].compara( x ) < 0 )
23         return busquedaBinaria( a, x, med + 1, sup );
24     else if( a[ med ].compara( x ) > 0 )
25         return busquedaBinaria ( a, x, inf, med - 1 );
26     else
27         return med;
28 }
```

Figura 7.9 Rutina de búsqueda binaria usando recursión.

En el método recursivo, el caso base de las líneas 19 y 20 trata el caso del vector vacío. En caso contrario, seguimos las descripción dada, haciendo una llamada recursiva sobre el fragmento del vector apropiado (líneas 23 o 25) si no se ha encontrado ya el elemento en el centro. Cuando se encuentra el elemento, se devuelve su posición en la línea 27.

Observe que el tiempo de ejecución, en términos de O, no cambia con respecto a la implementación no recursiva, ya que estamos realizando esencialmente el mismo trabajo. En la práctica, el tiempo de ejecución será ligeramente superior debido a los costes ocultos de la recursión.

Dibujando una regla

La Figura 7.10 muestra el resultado de la ejecución de un programa Java que dibuja las marcas de una regla. Aquí consideramos el problema de dibujar una pulgada. En el medio se coloca la marca más larga. En la Figura 7.10, tanto a la izquierda como a la derecha de la marca central encontramos una versión miniaturizada de la regla. Esto sugiere un algoritmo recursivo que dibuja primero la línea central y después las mitades izquierda y derecha.

El Apéndice D proporciona los detalles del dibujo de líneas y figuras en Java. Basta decir aquí que el programador debe proporcionar un método `paint`, el cual puede llamar a otros métodos. El método `pintaRegla` de la Figura 7.11 es la rutina recursiva llamada desde `paint`. Usa el método `drawLine`, que es parte de la librería de Java. Este método pinta una línea desde un punto de coordenadas (x,y)

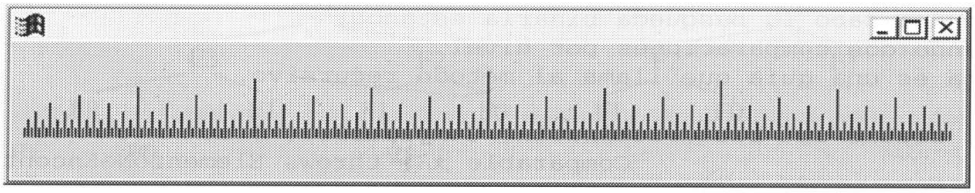

Figura 7.10 Una regla dibujada recursivamente.

```
1  private void
2  pintaRegla( Graphics g, int izdo, int dcho, int nivel)
3  {
4      if( nivel < 1 )
5          return;
6
7      int med = ( izdo + dcho ) / 2;
8
9      g.drawLine( med, 80, med, 80 - nivel * 5 );
10
11     pintaRegla( g, izdo, med - 1, nivel- 1 );
12     pintaRegla( g, med + 1, dcho, nivel - 1 );
13 }
```

Figura 7.11 Método recursivo para dibujar una regla.

a otro punto, donde las coordenadas son desplazamientos desde la esquina superior izquierda.

Nuestra rutina dibuja marcas a `nivel` alturas diferentes; cada llamada recursiva es un nivel más profunda (en la Figura 7.10 hay ocho niveles). En las líneas 4 y 5 se trata el caso base. En la línea 9 se dibuja la línea central y después se dibujan recursivamente las dos miniaturas en las líneas 11 y 12. En el código disponible en Internet, incluimos código extra para ralentizar el proceso de dibujo, de forma que se pueda ir viendo el orden en que se pintan las líneas.

Estrella fractal

En la parte izquierda de la Figura 7.12 aparece aparentemente un complejo patrón, llamado *estrella fractal*, el cual sin embargo, se puede dibujar fácilmente usando recursión. El dibujo entero es inicialmente gris (no se muestra en la figura). El patrón se forma dibujando cuadrados blancos sobre el fondo gris. En la parte derecha de la Figura 7.12 se muestra el dibujo antes de añadir el último cuadrado, que va en el centro. Parece evidente que antes de añadir el último cuadrado del centro, se ha tenido que dibujar cuatro versiones en miniatura, una para cada uno de los cuadrantes, lo cual proporciona la información necesaria para derivar el algoritmo recursivo.

Igual que en el ejemplo anterior, el método `pintaFractal` será llamado desde `paint` y usa una rutina de la librería de Java, en este caso `fillRect`, que pinta un rectángulo cuya esquina superior izquierda y dimensiones deben especificarse. En la Figura 7.13 se muestra el código resultante. Los parámetros de `pintaFractal` incluyen el centro del fractal y la dimensión total. Con ellos podemos calcular en la línea 5 el tamaño del cuadrado grande del centro. Después de tratar el caso base en las líneas 7 y 8, calculamos los límites del rectángulo central. Entonces podemos dibujar los cuatro pequeños fractales en las líneas 17 a 20. Finalmente dibujamos el cuadrado central en la línea 23. Observe que el cuadrado se debe pintar tras las llamadas recursivas, ya que en caso contrario se obtendría un dibujo diferente (en el Ejercicio 7.28 se le pide que explique la diferencia).

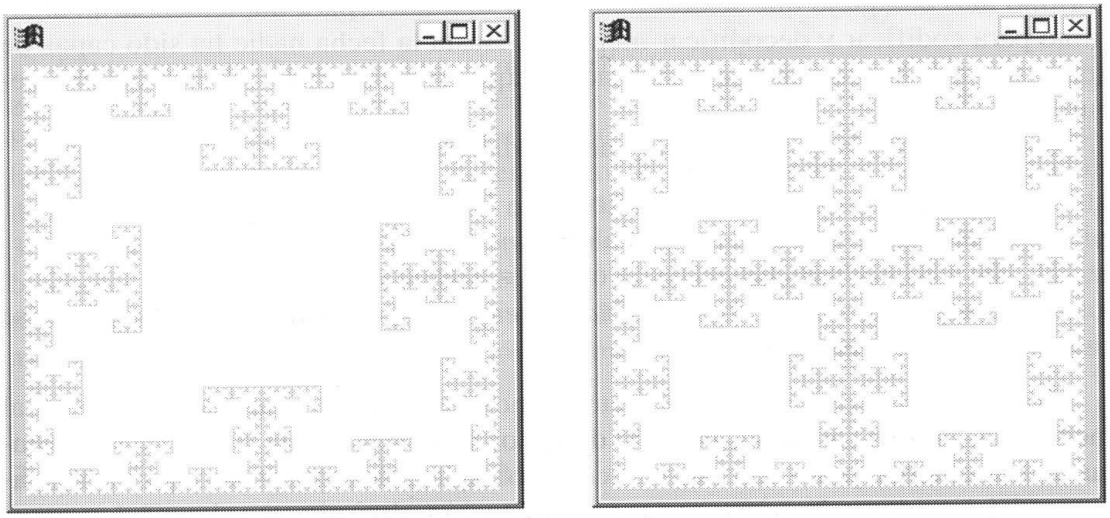

Figura 7.12 La figura de la izquierda es una estrella fractal dibujada por el programa de la Figura 7.13. La figura de la derecha es la misma estrella antes de añadir el último cuadrado.

```
1  // Pinta el dibujo de la Figura 7.12
2  private void pintaFractal( Graphics g, int xCentro,
3                             int yCentro, int cotaDim )
4  {
5      int lado = cotaDim / 2;
6
7      if( lado < 1 )
8          return;
9
10         // Cálculo de las esquinas
11     int izdo = xCentro - lado / 2;
12     int sup =  yCentro - lado / 2;
13     int dcho = xCentro + lado / 2;
14     int inf =  yCentro + lado / 2;
15
16         // Dibujo recursivo de los cuatro cuadrantes
17     pintaFractal( g, izdo, sup, cotaDim / 2 );
18     pintaFractal( g, izdo, inf, cotaDim / 2 );
19     pintaFractal( g, dcho, sup, cotaDim / 2 );
20     pintaFractal( g, dcho, inf, cotaDim / 2 );
21
22         // Dibuja el cuadro central, que se solapa con los cuadrantes
23     g.fillRect( izdo, sup, dcho - izdo, inf - sup );
24 }
```

Figura 7.13 Código para dibujar la estrella fractal de la Figura 7.12.

7.4 Aplicaciones numéricas

Esta sección estudia tres problemas que proceden de la teoría de números. La teoría de números se solía considerar una rama interesante pero inútil de las matemáticas. Pero, en los últimos 20 años, ha surgido una importante aplicación de la teoría de números: la seguridad de los datos. La discusión comienza con unos breves fundamentos matemáticos, para continuar con sendos algoritmos recursivos para resolver tres problemas. Podemos combinar estas rutinas, junto con un cuarto algoritmo más complejo que se describe en el Capítulo 9, para implementar un algoritmo para codificar y decodificar mensajes. Hasta la fecha nadie ha sido capaz de demostrar que el esquema de encriptación aquí descrito no es suficientemente seguro.

Estos son los cuatro problemas que estudiaremos:

1. *Exponenciación modular*: cálculo de $X^N (\text{mod } P)$.
2. *Máximo común divisor*: cálculo de $mcd(A, B)$.
3. *Inverso multiplicativo*: resolver $AX \equiv 1 (\text{mod } P)$ para X.
4. *Comprobación de la propiedad de ser primo*: determinar si N es primo (se pospone hasta el Capítulo 9).

Todos los enteros con los que pretendemos trabajar son grandes y requieren al menos 100 dígitos cada uno. Por ello debemos disponer de una forma de representar dichos enteros, junto con una colección completa de algoritmos para las operaciones básicas como la suma, resta, multiplicación, división, etc. Java 1.1 proporciona para este propósito una clase denominada `BigDecimal`. Implementarla de forma eficiente no es una tarea trivial, y de hecho existe una gran cantidad de literatura sobre el tema.

Los algoritmos aquí descritos tienen la propiedad de que cuando trabajan con ese tipo de objetos aún requieren una cantidad razonable de tiempo. Sin embargo, los métodos que escribimos funcionarán con el tipo `long`, de modo que podamos fácilmente comprobar el funcionamiento de los algoritmos básicos. Debido a que Java no permite la sobrecarga de operadores, sería necesario rescribir el código usando métodos con nombre, en lugar de operadores, si queremos usar objetos `BigDecimal`.

7.4.1 Aritmética modular

En los distintos problemas que abordamos en esta sección, al igual que sucedía en la implementación de la tabla hash de la Sección 6.7 y del Capítulo 19, se requiere el uso del operador de módulo en Java, el cual se denota mediante el operador `%`. Este operador calcula el resto de la división entre dos valores de tipo entero. Por ejemplo, `13%10` se evalúa a 3, al igual que `3%10` y `23%10`. Cuando calculamos el resto de una división por 10, el rango de posibles resultados está entre 0 y 9. Esto hace que el operador `%` sea útil cuando necesitamos generar enteros pequeños.

Si dos números A y B producen el mismo resto al dividirlos por N, decimos que son congruentes módulo N, lo que se denota por $A \equiv B(\text{mod } N)$. En tal caso, se cumplirá que N divide a $A - B$. Además, el recíproco también es cierto: si N divide a $A - B$, entonces $A \equiv B(\text{mod } N)$. Puesto que sólo hay N posibles restos — 0, 1, ..., $N - 1$ — decimos que los enteros se dividen en N clases de congruencia módulo N. En otras palabras, cada entero se puede colocar en una de las N clases, y aquellos que pertenecen a la misma clase son congruentes entre sí, módulo N. Hay tres propiedades importantes que usamos en nuestros algoritmos (dejamos su demostración como Ejercicio 7.10):

1. Si $A \equiv B(\text{mod } N)$, entonces para cualquier C, $A + C \equiv B + C(\text{mod } N)$.
2. Si $A \equiv B(\text{mod } N)$, entonces para cualquier D, $AD \equiv BD(\text{mod } N)$.
3. Si $A \equiv B(\text{mod } N)$, entonces para cualquier P, $A^P \equiv B^P(\text{mod } N)$.

Estos teoremas permiten que algunos cálculos se hagan con un menor esfuerzo. Por ejemplo, supongamos que queremos conocer el último dígito de 3333^{5555}. Puesto que este número tiene más de 15.000 dígitos, es demasiado costoso calcular directamente la respuesta. Sin embargo, lo que queremos calcular en realidad es $3333^{5555}(\text{mod } 10)$. Puesto que $3333 \equiv 3(\text{mod } 10)$, basta con calcular $3^{5555}(\text{mod } 10)$. Puesto que $3^4 = 81$, sabemos que $3^4 \equiv 1(\text{mod } 10)$, y elevando ambos lados a la potencia 1.388 obtenemos $3^{5552} \equiv 1(\text{mod } 10)$. Si ahora multiplicamos ambos lados por $3^3 = 27$, obtenemos $3^{5555} \equiv 27 \equiv 7(\text{mod } 10)$, completando así el cálculo. En la siguiente sección se muestra cómo generalizar este proceso.

7.4.2 Exponenciación modular

En esta sección se muestra cómo calcular de forma eficiente $X^N(\text{mod } P)$. Esto se puede hacer inicializando el resultado `tmp` a 1 y después multiplicando repetidamente el resultado `tmp` por X y aplicando el operador `%` tras cada multiplicación.

Aplicar el operador % de esta forma, en lugar de hacerlo después de la última multiplicación, hace que las multiplicaciones sean más sencillas, ya que `resultado tmp` se mantiene pequeño.

Tras N multiplicaciones, `resultado tmp` es la respuesta que estamos buscando. Sin embargo, si N es un número de 100 dígitos, la multiplicación es aún impracticable. De hecho, si N vale mil millones, resulta impracticable en cualquier máquina excepto en las más rápidas.

Un algoritmo más rápido se basa en la siguiente observación: si N es par entonces $X^N = (X \cdot X)^{\lfloor N/2 \rfloor}$ y si es impar entonces $X^N = X \cdot X^{N-1} = X \cdot (X \cdot X)^{\lfloor N/2 \rfloor}$. (Recuerde que $\lfloor X \rfloor$ es el mayor entero que es menor o igual que X.) Al igual que antes, aplique el operador % tras cada multiplicación.

El algoritmo recursivo de la Figura 7.14 representa una implementación directa de la estrategia. En las líneas 7 y 8 se trata el caso base: X^0 vale 1 por definición[4]. En la línea 10, hacemos una llamada recursiva basada en la primera igualdad del párrafo anterior. Si N es par, dicha fórmula calcula la respuesta deseada, y si es impar necesitamos multiplicarla por una N extra (usando después el operador %).

La exponenciación se puede realizar con un número logarítmico de multiplicaciones.

Este algoritmo es más rápido que el algoritmo más simple propuesto anteriormente. Si $M(N)$ es el número de multiplicaciones usadas por `potencia`, entonces $M(N) \leqslant M\lfloor N/2 \rfloor + 2$; pues si N es par, llevamos a cabo una multiplicación más las que se generan recursivamente, mientras que si N es impar, realizamos dos multiplicaciones más las que se ejecutan recursivamente. Puesto que $M(0) = 0$, podemos demostrar que $M(N) < 2 \log N$. El factor logarítmico se puede obtener sin necesidad de un cálculo directo aplicando el principio de sucesivas divisiones por la mitad (*véase* Sección 5.5), que nos da el número de llamadas recursivas a `potencia`. Además, el valor medio de $M(N)$ es $(3/2) \log N$, ya que en cada paso recursivo, es igualmente probable que N sea par o impar. Si N es un número de 100 dígitos, entonces, en el peor caso, sólo se necesitan 655 multiplicaciones (y sólo 500 en media).

```
1  /**
2   * Devuelve x^n (mod p)
3   * Suponiendo que p es positivo y potencia(0, 0, p) es 1
4   */
5  public static long potencia( long x, long n, long p )
6  {
7      if( n == 0 )
8          return 1;
9
10     long tmp = potencia( ( x * x ) % p, n / 2, p );
11     if( n % 2 != 0 )
12         tmp = ( tmp * x ) % p;
13
14     return tmp;
15 }
```

Figura 7.14 Rutina de exponenciación modular.

[4] Para los propósitos de este algoritmo, $0^0 = 1$, N es no negativo y P es positivo.

7.4.3 Máximo común divisor e inversos multiplicativos

El *máximo común divisor* (mcd) de dos enteros es el mayor entero que divide a ambos.

Considere dos enteros no negativos A y B. Su máximo común divisor, $mcd(A, B)$, es el mayor entero D que divide a A y a B. Por ejemplo, $mcd(70, 25)$ es 5.

Es fácil verificar que $mcd(A, B) \equiv mcd(A - B, B)$. Si D divide a A y a B, también debe dividir a $A - B$; y si D divide a $A - B$ y a B, entonces también debe dividir a A.

Esta observación nos lleva a un algoritmo simple en el que restamos repetidamente B a A, transformando así el problema en uno más pequeño. Eventualmente A se hará más pequeño que B, y entonces podemos intercambiar los papeles de A y B y continuar a partir de ahí. En algún punto, B se hará 0, y entonces, como sabemos que $mcd(A, 0) \equiv A$, y puesto que cada transformación preserva el *mcd* de los A y B originales, ya tenemos nuestra respuesta. A este algoritmo se le llama algoritmo de Euclides y se describió por primera vez hace 2.000 años. Aunque es correcto, no se puede usar para objetos de la clase `BigDecimal` porque se requeriría un enorme número de restas.

Una modificación que hace más eficiente al algoritmo se basa en el hecho de que restar repetidamente B a A hasta que A es más pequeño que B, es equivalente a convertir A en $A \bmod B$; es decir $mcd(A, B) \equiv mcd(B, A \bmod B)$. Esta definición recursiva, junto con el caso base, en el que $B = 0$, se usa directamente para obtener la rutina de la Figura 7.15. Para ver cómo funciona, observe que en el ejemplo anterior se produciría la siguiente secuencia de llamadas recursivas para deducir que el *mcd* de 70 y 25 es 5: $mcd(70, 25) \Rightarrow mcd(25, 20) \Rightarrow mcd(20, 5) \Rightarrow mcd(5, 0) \Rightarrow 5$.

El número de llamadas recursivas es proporcional al logaritmo de A, es decir, es del mismo orden de magnitud que el resto de rutinas estudiadas en esta sección. Esto se debe a que al hacer una llamada recursiva el argumento se reduce al menos a la mitad. La demostración se deja como Ejercicio 7.11.

El mcd y el inverso multiplicativo se pueden calcular en un tiempo logarítmico usando una variante del algoritmo de Euclides.

El algoritmo de *mcd* se usa de forma implícita para resolver un problema matemático similar. La solución $1 \leqslant X < N$ a la ecuación $AX \equiv 1 \pmod N$ se llama *inverso multiplicativo* de A, módulo N. Supongamos también que $1 \leqslant A < N$. Como ejemplo, el inverso de 3 módulo 13 es 9: $3 \cdot 9 \bmod 13$ devuelve 1.

La posibilidad de calcular inversos multiplicativos es importante porque ecuaciones como $3i \equiv 7 (\text{mod } 13)$ se resuelven fácilmente si conocemos el inverso multiplicativo. Este tipo de ecuaciones surgen en muchas aplicaciones, incluyendo el algoritmo de encriptación estudiado al final de esta sección. En el ejemplo ante-

```
1  /**
2   * Devuelve el máximo común divisor.
3   */
4  public static long mcd( long a, long b )
5  {
6      if( b == 0 )
7          return a;
8      else
9          return mcd(b, a % b);
10 }
```

Figura 7.15 Cálculo del máximo común divisor.

rior, si multiplicamos por el inverso de 3 (es decir, 9), obtenemos $i \equiv 63(\text{mod } 13)$, por lo que $i = 11$ es una solución. Si

$$AX \equiv 1(\text{mod } N)$$

entonces

$$AX + NY \equiv 1(\text{mod } N)$$

es cierto para cualquier Y. Para algún Y, el lado izquierdo debe ser exactamente 1. Luego la ecuación

$$AX + NY = 1$$

tiene solución si y sólo si A tiene un inverso multiplicativo.

Dados A y B, mostramos cómo encontrar X e Y que cumplan

$$AX + BY = 1.$$

Supondremos, sin pérdida de generalidad, que $0 \leqslant |B| < |A|$. Extenderemos el algoritmo del *mcd* para calcular X e Y.

Consideremos primero el caso base $B \equiv 0$, para el cual tenemos que resolver $AX = 1$, lo que implica que tanto A como X valen 1, ya que si A no es 1, entonces no hay inverso multiplicativo. Una consecuencia de esto es que A tiene inverso multiplicativo módulo N si y sólo si $mcd(A, N) = 1$.

Si B no es 0, recuerde que $mcd(A, B) = mcd(B, A \text{ mod } B)$. Sea $A = BQ + R$, donde Q es el cociente y R es el resto. La llamada recursiva es entonces $mcd(B, R)$. Supongamos que podemos resolver recursivamente

$$BX_1 + RY_1 = 1.$$

Puesto que $R = A - BQ$, tenemos

$$BX_1 + (A - BQ)Y_1 = 1,$$

lo que significa que

$$AY_1 + B(X_1 - QY_1) = 1.$$

Luego $X = Y_1$ y $Y = X_1 = \lfloor A/B \rfloor Y_1$ es una solución de $AX + BY = 1$. La codificación de este algoritmo aparece en la Figura 7.16 con el nombre de `mcdCompleto`. La rutina `inverso` simplemente llama a `mcdCompleto`, donde X e Y son variables estáticas. El único detalle pendiente es que el valor de X podría ser negativo, en cuyo caso la línea 35 de `inverso` lo convertirá en positivo. La demostración de la corrección se puede hacer por inducción.

```
1       // Variables internas de mcdCompleto
2   private static long x;
3   private static long y;
4
5   /**
6    * Sigue hacia atrás el algoritmo de Euclides para encontrar
7    * x e y tales que mcd(a, b) = 1,
8    * ax + by = 1,
9    */
10  private static void mcdCompleto( long a, long b )
11  {
12      long x1, y1;
13
14      if( b == 0)
15      {
16          x = 1;          // Si a != 1, no hay inverso
17          y = 0;          // Omitimos esta comprobación
18      }
19      else
20      {
21          mcdCompleto( b, a % b );
22          x1 = x; y1 = y;
23          x = y1;
24          y = x1 - (a / b) * y1;
25      }
26  }
27
28  /**
29   * Resuelve ax == 1 (mod n), suponiendo que mcd( a, n ) = 1.
30   * @return x.
31   */
32  public static long inverso( long a, long n )
33  {
34      mcdCompleto( a, n );
35      return x > 0 ? x : x + n;
36  }
```

Figura 7.16 Rutina para determinar el inverso multiplicativo.

7.4.4 El sistema de criptografía RSA

La teoría de números se usa en criptografía porque el proceso de factorizar un número es mucho más costoso que la multiplicación.

Durante siglos se pensó que la teoría de números era una rama de las matemáticas completamente inútil desde el punto de vista práctico, pero recientemente se ha convertido en un campo importante debido a sus aplicaciones a la criptografía.

El problema que consideramos es el siguiente: supongamos que Alicia quiere enviar un mensaje a Roberto, pero que a ella le preocupa que la transmisión sea interceptada. Por ejemplo, si la transmisión se realiza a través de una línea telefónica y el teléfono está pinchado, entonces alguien más puede estar escuchando el mensaje. Supondremos, eso sí, que incluso si hay escuchas en la línea telefónica, no se produce ningún tipo de daño sobre la señal; es decir, Roberto obtiene exactamente lo que le envía Alicia.

La *encriptación* se usa para transmitir mensajes de forma que nadie más que el emisor y el receptor puedan entenderlos.

El *sistema de encriptación RSA* es un método de encriptación bastante popular.

Una solución a este problema es usar un esquema de encriptación que consta de dos partes. Primero, Alicia *codifica* el mensaje y envía el resultado, el cual ya no se puede leer directamente. Cuando Roberto recibe la transmisión de Alicia, la *decodifica* para obtener el mensaje original. La seguridad del algoritmo se basa en que nadie más que Roberto podría llevar a cabo la decodificación del mensaje (ni siquiera Alicia, si no ha guardado el original).

Para aplicar el proceso, Roberto debe proporcionar a Alicia un método de encriptación que solamente él sepa cómo invertir. Este problema es un serio reto. Muchos de los algoritmos que han venido siendo propuestos pueden verse comprometidos por alguna técnica sutil para averiguar el código. Aquí se describe un método, llamado *sistema de encriptación RSA* (por las iniciales de sus autores) que es una implementación muy elegante de una estrategia de encriptación.

Nuestro objetivo es dar solamente una visión de alto nivel, mostrando cómo interactúan los métodos escritos en esta sección. Las referencias contienen punteros a descripciones más detalladas, así como a demostraciones de las propiedades clave de los algoritmos.

Observemos primero que un mensaje consta de una secuencia de caracteres y que cada carácter es una secuencia de bits; luego un mensaje es, al fin y al cabo, una secuencia de bits. Si partimos el mensaje en bloques de B bits, podemos interpretar el mensaje como una serie de números muy grandes, por lo que el problema básico se reduce a codificar y después decodificar números grandes.

Cálculo de las constantes RSA

El algoritmo RSA comienza exigiendo que el receptor del mensaje determine ciertas constantes adecuadas. Primero, se eligen aleatoriamente dos números primos grandes p y q, habitualmente de 100 dígitos o más cada uno. Para hacer manejable el ejemplo, tomaremos $p = 127$ y $q = 211$. Observe que Roberto, que es el receptor, es quien está haciendo este proceso. Como quiera que hay muchos números primos, Roberto puede continuar eligiendo pares de números primos hasta que uno de ellos cumpla el test de primalidad, que se discutirá en el Capítulo 9.

A continuación, Roberto calcula $N = pq$ y $N' = (p - 1)(q - 1)$. En nuestro ejemplo, obtenemos $N = 26.797$ y $N' = 26.460$. Roberto continúa eligiendo un $e > 1$ tal que $mcd(e, N') = 1$. En términos matemáticos, elige cualquier e que sea relativamente primo con N'. De nuevo, Roberto puede ir probando con distintos valores de e, usando la rutina de la Figura 7.15, hasta encontrar uno que satisfaga la propiedad. En particular, como cualquier primo e valdría, encontrar e es tan fácil como encontrar un número primo. En nuestro caso, $e = 13.379$ es una de las muchas opciones. A continuación, se calcula el inverso multiplicativo de e módulo N', d, usando la rutina de la Figura 7.16. En el ejemplo, $d = 11.099$.

Una vez que Roberto ha calculado todas estas constantes, hace lo siguiente: primero, destruye p, q y N'. La seguridad del sistema se podría ver comprometida si se cuenta con dichos valores. Entonces, Roberto le dice a todo el mundo que quiera enviarle un mensaje codificado cuáles son los valores de e y N, aunque guarda en secreto el valor de d.

Algoritmos de codificación y decodificación

Para codificar un entero M, el emisor calcula $M^e \pmod N$ y lo envía. En nuestro caso $M = 10.237$, luego el valor enviado es 8.422. Cuando se recibe un entero codificado R, todo lo que tiene que hacer Roberto es calcular $R^d \pmod N$. Para $R = 8.422$ se puede comprobar que se obtiene de nuevo el $M = 10.237$ original, lo cual no es en absoluto accidental. Tanto la codificación como la decodificación se pueden llevar a cabo usando la rutina de exponenciación modular de la Figura 7.14.

¿Por qué funciona el algoritmo? La elección de e, d y N garantiza (por medio de una demostración en teoría de números fuera del ámbito de este libro) que $M^{ed} \equiv M \pmod N$, siempre que N y M no tengan factores comunes. Puesto que los únicos factores de N son dos números primos de 100 dígitos, es prácticamente imposible que esto suceda[5]. Por tanto, la decodificación del texto codificado devuelve siempre el texto original.

Lo que hace que el sistema que hemos visto sea seguro es que en principio se necesita d para poder decodificar un mensaje. Ahora bien N y e determinan únicamente a d. Por ejemplo, si factorizamos N, obtenemos p y q y a partir de ellos se puede reconstruir d. El problema está en que factorizar números grandes es aparentemente muy difícil. En consecuencia, la seguridad del sistema RSA se basa en el hecho de que se piensa que factorizar números muy grandes es intrínsecamente muy difícil. Como quiera que hasta la fecha nadie ha sido capaz de factorizar de forma eficiente, aunque tampoco tenemos la prueba de que ello sea un hecho imposible, hasta el momento el método funciona bien.

Este esquema general recibe el nombre de *criptografía de clave pública*. Cualquiera que quiera recibir mensajes, publica información de codificación para todos aquellos que quieran usarla. En el sistema RSA, e y N se calcularían una vez por parte de cada receptor y se harían públicos en una lista.

En la *criptografía de clave pública*, cada participante publica el código que otros pueden usar para enviarle mensajes codificados, pero se guarda su código secreto para decodificar los mensajes recibidos.

El algoritmo RSA se usa ampliamente para implementar correo seguro, así como transacciones seguras en Internet. Cuando vea un candado cerrado (🔒) en la parte inferior de una página Web en Netscape Navigator, se está realizando una transacción segura usando criptografía. El método que se emplea realmente es más complejo que el aquí descrito. La razón que justifica dicho cambio de método es que el algoritmo RSA resulta lento cuando se trata de enviar mensajes bastante largos.

Un método más rápido es el llamado *DES*. A diferencia del algoritmo RSA, DES es un algoritmo de clave única, lo que significa que la misma clave sirve para codificar y decodificar. Esto es semejante al típico cerrojo en la puerta de nuestra casa. El problema con la clave única es que ambas partes tienen que compartir la clave única. ¿Cómo se asegura una parte de que la otra tiene la clave única? Esto se resuelve usando el algoritmo RSA. Una solución habitual es que, Alicia genera aleatoriamente una clave para la codificación DES. Con ella codifica el mensaje usando DES, lo cual es mucho más rápido que usar RSA. Transmite el mensaje codificado a Roberto. Para que Roberto pueda decodificar el mensaje, primero necesita obtener la clave DES. Puesto que dicha clave es relativamente corta, Alicia

En la práctica, el algoritmo RSA se usa para codificar la clave usada en un algoritmo de encriptación de clave única, como el DES.

[5] Es más probable que gane la lotería primitiva 13 semanas seguidas. Sin embargo, si esto ocurriera, el sistema estaría comprometido, porque el *mcd* será un factor de N.

puede usar RSA para codificar la clave DES, enviándola en una segunda transmisión a Roberto. Roberto decodifica la segunda transmisión, obteniendo así la clave DES, con la que se puede decodificar el mensaje original. Este tipo de protocolos con refuerzos forman la base de la mayor parte de las implementaciones prácticas de la codificación.

7.5 Algoritmos divide y vencerás

Los algoritmos divide y vencerás son algoritmos recursivos generalmente muy eficientes.

Una técnica importante de resolución de problemas que hace uso de la recursión es la técnica de *divide y vencerás*. Los algoritmos divide y vencerás son algoritmos recursivos que constan de dos partes:

- *Dividir*: se resuelven recursivamente problemas más pequeños (excepto, por supuesto, los casos base).
- *Vencer*: la solución al problema original se consigue a partir de las soluciones a los subproblemas.

Tradicionalmente, las rutinas en las que el algoritmo contiene al menos dos llamadas recursivas se llaman algoritmos divide y vencerás, al contrario que las rutinas cuyo texto contiene solamente una llamada recursiva. En consecuencia, las rutinas recursivas vistas hasta el momento en este capítulo, no son algoritmos divide y vencerás. Además, los subproblemas deben ser disjuntos (más exactamente, esencialmente sin superposiciones) para evitar los costes excesivos que ya se vieron en el cálculo recursivo de los números de Fibonacci. En esta sección presentaremos un ejemplo del paradigma divide y vencerás: veremos cómo usar la recursión para resolver el problema de la obtención de la subsecuencia de suma máxima. A continuación presentamos el análisis de esta solución recursiva, demostrando que su tiempo de ejecución es $O(N \log N)$. Aunque ya vimos un algoritmo lineal para resolver este problema, lo que presentamos ahora es un ejemplo representativo de una gran variedad de aplicaciones, incluyendo los algoritmos de ordenación, como mergesort y quicksort, estudiados en el Capítulo 8. Resulta por tanto muy importante estudiar esta técnica. Para terminar, discutiremos la forma general del tiempo de ejecución de una amplia variedad de algoritmos divide y vencerás.

7.5.1 El problema de la subsecuencia de suma máxima

Recordemos el problema de encontrar, en una secuencia de números, una subsecuencia contigua de suma máxima, que estudiamos en la Sección 5.3. Para facilitar la lectura, volvemos a incluir su enunciado a continuación:

PROBLEMA DE LA OBTENCIÓN DE LA SUBSECUENCIA DE SUMA MÁXIMA

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

El problema de la obtención de la subsecuencia contigua de suma máxima se puede resolver usando un algoritmo divide y vencerás.

Para resolver este problema se presentaron tres algoritmos de distintas complejidades. Uno de ellos era un algoritmo cúbico basado en una búsqueda exhaustiva: calculábamos la suma de cada posible subsecuencia y elegíamos la máxima. Se describió también una mejora con tiempo cuadrático que aprovechaba el hecho de que la suma de una nueva subsecuencia se puede calcular en tiempo constante a partir de la anterior. Puesto que tenemos $O(N^2)$ subsecuencias, ésta es la mejor cota que se puede conseguir usando una aproximación que examine directamente todas las subsecuencias. Vimos, por último, un algoritmo con coste en tiempo lineal que examinaba solamente unas pocas subsecuencias, y cuya corrección no era obvia.

Diseñaremos ahora un algoritmo divide y vencerás. Supongamos que la sucesión dada es $\{4, -3, 5, -2, -1, 2, 6, -2\}$. Dividiremos esta entrada en dos partes iguales, tal y como se muestra en la Figura 7.17. Entonces la subsecuencia de suma máxima puede aparecer en una de estas tres formas:

- *Caso* 1: está totalmente incluida en la primera mitad.
- *Caso* 2: está totalmente incluida en la segunda mitad.
- *Caso* 3: comienza en la primera mitad, pero termina en la segunda.

Mostraremos cómo encontrar los máximos valores de cada uno de estos tres casos, de una forma más eficiente que la que correspondería a una búsqueda exhaustiva.

Empezamos por el caso 3. Queremos evitar el bucle anidado generado al considerar independientemente todos los $N/2$ puntos de comienzo y los $N/2$ puntos de finalización. La idea es sustituir dos bucles anidados por dos bucles consecutivos. Los bucles consecutivos, cada uno de tamaño $N/2$, se combinan, requiriendo en total un tiempo lineal. Podemos hacer esto porque cualquier subsecuencia contigua que comience en la primera mitad y termine en la segunda, debe incluir el último elemento de la primera mitad y el primero de la segunda mitad.

La Figura 7.17 muestra que podemos calcular, para cada elemento de la primera mitad, la suma de la subsecuencia contigua que termina en el elemento situado más a la derecha. Hacemos esto con un recorrido de derecha a izquierda, partiendo del elemento situado entre las dos mitades. Análogamente, podemos calcular la suma de todas las subsecuencias contiguas que comiencen con el primer elemento de la segunda mitad. Entonces se puede combinar estas dos subsecuencias para formar la subsecuencia de suma máxima que cruza la línea divisoria. En el ejemplo de la Figura 7.17, la secuencia resultante va desde el primer elemento de la primera mitad hasta el penúltimo elemento de la segunda mitad. La suma total es la suma de las dos subsecuencias, $4 + 7 = 11$.

Primera mitad				Segunda mitad				
4	−3	5	−2	−1	2	6	−2	Valores
4*	0	3	−2	−1	1	7*	5	Sumas

Suma desde el centro
(*denota el máximo para cada mitad)

Figura 7.17 Dividiendo en dos el problema de la obtención de la subsecuencia máxima.

En el método divide y vencerás, la recursión es el «divide», la sobrecarga es el «vencerás».

Este análisis nos muestra que el caso 3 se puede resolver en tiempo lineal. Pero, ¿qué pasa con los casos 1 y 2? Puesto que hay $N/2$ elementos en cada mitad, la aplicación de una búsqueda exhaustiva en cada mitad requeriría un tiempo cuadrático por cada mitad. Específicamente, lo único que hemos hecho es eliminar la mitad del trabajo, y la mitad de un tiempo cuadrático sigue siendo cuadrático. Pero podemos aplicar la misma estrategia de división por la mitad en los casos 1 y 2. Podemos continuar dividiendo hasta que sea imposible dividir más. Esto equivale, más concretamente, a resolver los casos 1 y 2 recursivamente. Como se demostrará más adelante en el capítulo, esto reducirá el tiempo de ejecución por debajo de cuadrático, pues los ahorros se acumulan a lo largo de la ejecución del algoritmo. Mostramos a continuación un esquema del algoritmo:

1. Calcular recursivamente la subsecuencia de suma máxima que está totalmente contenida en la primera mitad.
2. Calcular recursivamente la subsecuencia de suma máxima que está totalmente contenida en la segunda mitad.
3. Calcular, usando dos bucles consecutivos, la subsecuencia de suma máxima que comienza en la primera mitad pero termina en la segunda.
4. Elegir la mayor de las tres sumas.

El método Java resultante aparece en la Figura 7.18. Un algoritmo recursivo nos exige definir un caso base. Naturalmente, cuando el dato es un solo elemento, no usamos recursión.

A la llamada recursiva se le pasa el vector de entrada junto con los límites izquierdo y derecho, los cuales delimitan la porción de vector sobre la que se está operando. Una rutina guía de una línea inicializa los parámetros límite a 0 y $N-1$.

Las líneas 13 y 14 tratan el caso base. Si `izdo==dcho`, sólo tenemos un elemento que conforma la subsecuencia de suma máxima si es no negativo; en caso contrario, la secuencia vacía con suma 0 es máxima. Las líneas 16 y 17 llevan a cabo las llamadas recursivas. Podemos ver que las llamadas recursivas siempre se realizan sobre un argumento más pequeño que el original, por lo que hacemos progresos hacia el caso base. Las líneas 19 a 24 y 25 a 30 calculan las sumas máximas que tocan el borde central. La suma de estos dos valores es la suma máxima que se extiende por ambas mitades. La rutina trivial `max3` (que no se muestra) devuelve la mayor de las tres soluciones parciales.

7.5.2 Análisis de un algoritmo divide y vencerás sencillo

Un análisis informal del algoritmo divide y vencerás para el cálculo de la subsecuencia contigua de suma máxima nos dice que gastamos un tiempo $O(N)$ por nivel.

El algoritmo recursivo para la obtención de la subsecuencia de suma máxima usa un tiempo lineal para calcular la suma de la mayor secuencia que atraviesa el borde, y después hace dos llamadas recursivas. Las llamadas recursivas calculan una suma que atraviesa el borde central, después hacen más llamadas recursivas y así sucesivamente. El trabajo total invertido por el algoritmo es entonces proporcional al coste de las búsquedas realizadas por todas las llamadas recursivas.

La Figura 7.19 ilustra gráficamente cómo trabaja el algoritmo para $N = 8$ elementos. Cada rectángulo representa una llamada a `maxSumRec`, y la longitud del rectángulo es proporcional al tamaño del subvector (y por tanto al coste del recorrido del subvector) sobre el que actúa la llamada. La llamada inicial se muestra en la primera línea. Observe que el tamaño del subvector es N, lo cual representa

```
1  /**
2   * Algoritmo de obtención de la subsecuencia de suma máxima.
3   * Encuentra la suma máxima de un subvector a[izdo..dcho].
4   * No intenta mantener la mejor secuencia.
5   */
6  private static int maxSumRec( int [ ] a, int izdo,
7                                            int dcho )
8  {
9      int maxSumIzdaBorde = 0, maxSumDchaBorde = 0;
10     int sumIzdaBorde = 0, sumDchaBorde = 0;
11     int centro = ( izdo + dcho ) / 2;
12
13     if( izdo == dcho )   //Caso base
14         return a[ izdo ] > 0 ? a[ izdo ] : 0;
15
16     int maxSumIzda = maxSumRec( a, izdo, centro );
17     int maxSumDcha = maxSumRec( a, centro + 1, dcho );
18
19     for( int i = centro; i >= izdo; i-- )
20     {
21         sumIzdaBorde += a[ i ];
22         if (sumIzdaBorde > maxSumIzdaBorde)
23             maxSumIzdaBorde = sumIzdaBorde;
24     }
25     for( int j = centro + 1; j <= dcho; j++ )
26     {
27         sumDchaBorde += a[ j ];
28         if (sumDchaBorde > maxSumDchaBorde)
29             maxSumDchaBorde = sumDchaBorde;
30     }
31
32     return max3( maxSumIzda, maxSumDcha,
33                  maxSumIzdaBorde + maxSumDchaBorde );
34 }
35
36 // Rutina visible públicamente
37 public static int maxSubSum4( int [ ] a)
38 {
39     return maxSumRec( a, 0, a.length - 1 );
40 }
```

Figura 7.18 Algoritmo divide y vencerás para el problema de la obtención de la subsecuencia de suma máxima.

el coste del recorrido para el tercer caso. La llamada inicial hace entonces dos llamadas recursivas sobre dos subvectores de tamaño $N/2$. El coste de cada recorrido en el caso 3 es la mitad del coste original, pero puesto que hay dos llamadas recursivas, el coste combinado de las dos llamadas recursivas es también N. Cada una de esas dos instancias recursivas hacen a su vez dos llamadas recursivas, generando por tanto cuatro subproblemas que son una cuarta parte del problema original. Luego el total de los costes para el caso 3 es también N.

Eventualmente, llegaremos al caso base. Cada uno de ellos tiene tamaño 1, pero tenemos N. Por supuesto, no hay costes para el caso 3 en este caso, pero consideramos que la comprobación que determina si el único elemento es positivo o negativo tiene un coste de 1 unidad de tiempo. Por tanto, el coste total por nivel de recursión, mostrado en la Figura 7.19, es N. Cada nivel reduce a la mitad el tamaño

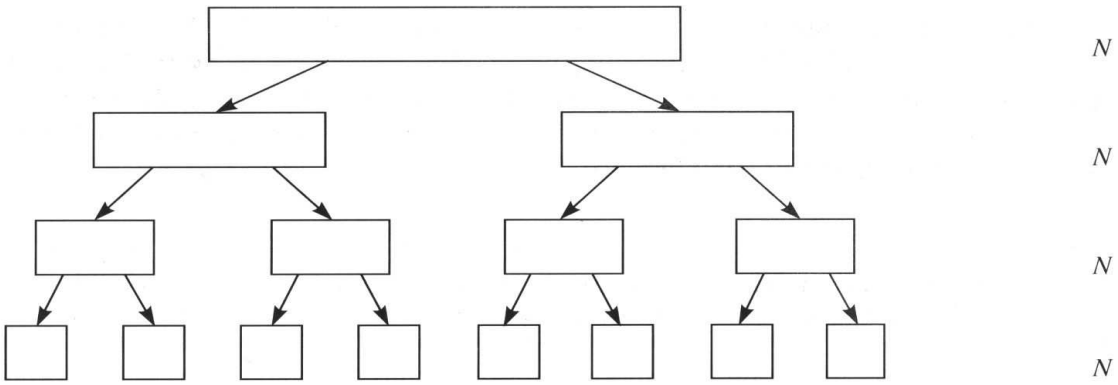

Figura 7.19 Traza de las llamadas recursivas del algoritmo de la subsecuencia contigua de suma máxima; $N = 8$ elementos.

del problema original, luego por el principio de las sucesivas divisiones por la mitad tenemos aproximadamente log N niveles. De hecho, el número de niveles es $1 + \lceil \log N \rceil$ (que vale 4 cuando N es igual a 8). Por tanto, el tiempo total de ejecución que esperamos es $O(N \log N)$.

Este análisis da una explicación intuitiva de por qué el tiempo de ejecución es $O(N \log N)$. Pero, en general, expandir un algoritmo recursivo para examinar su comportamiento es una mala idea, ya que viola la tercera regla de la recursión. A continuación presentaremos un tratamiento matemático más sistemático.

Un análisis más sistemático. Observe cuidadosamente que el razonamiento es válido para todas las clases de algoritmos que resuelven recursivamente dos mitades y usan un tiempo lineal adicional.

Sea $T(N)$ el tiempo que nos lleva resolver el problema de la obtención de la subsecuencia de suma máxima para un vector de tamaño N. Si $N = 1$, el programa tarda una cantidad de tiempo constante en ejecutar las líneas 13 a 14, la cual tomaremos como una unidad de tiempo. Luego $T(1) = 1$. En caso contrario, el programa debe hacer dos llamadas recursivas además del tiempo lineal que tarda en calcular la suma máxima para el caso 3. La sobrecarga constante es absorbida por el término $O(N)$. ¿Cuánto tardan las llamadas recursivas? Puesto que resuelven problemas de tamaño $N/2$, tenemos que cada una de ellas tarda $T(N/2)$ unidades de tiempo; por lo que el trabajo recursivo total es $2T(N/2)$. Esto genera las ecuaciones

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

Por supuesto, para que la segunda ecuación tenga sentido exacto, N debe ser una potencia de dos. En otro caso, en algún momento $N/2$ no será par. Una ecuación más precisa, válida para todo N, sería

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

No obstante, para simplificar los cálculos, supondremos que N es una potencia de 2 y reemplazamos el término $O(N)$ por N. Estas suposiciones son menores y no afectan al resultado O. En consecuencia, suponiendo que N es una potencia de 2, necesitamos obtener una solución explícita de $T(N)$ para

$$T(1) = 1,$$
$$T(N) = 2T(N/2) + N. \qquad \textbf{(7.6)}$$

Ésta es exactamente la ecuación que se ilustra en la Figura 7.19, luego ya sabemos que la respuesta ha de ser $N \log N + N$. Esto se puede ratificar fácilmente examinando unos pocos valores: $T(1) = 1$, $T(2) = 4$, $T(4) = 12$, $T(8) = 32$ y $T(16) = 80$. A continuación demostramos formalmente este resultado (Teorema 7.4) usando dos métodos diferentes.

Teorema 7.4

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

Demostración (Método 1)

Para N suficientemente grande, tenemos que $T(N/2) = 2T(N/4) + N/2$ por lo que podemos usar 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 usamos la Ecuación 7.6 para $N/4$ y multiplicamos por 4, obtenemos

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

que podemos sustituir en el lado derecho de la Ecuación 7.7 para obtener

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

Continuando de esta forma, obtenemos

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

Finalmente, tomando $k = \log N$ (lo cual tiene sentido, puesto que $2^k = N$), obtenemos

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

Aunque este método de prueba parece funcionar bien, puede ser difícil aplicarlo en casos más complicados, pues tiende a producir ecuaciones muy largas. A continuación se presenta un segundo método que parece más sencillo, porque genera ecuaciones verticalmente, que se manipulan más fácilmente.

Demostración (Método 2)

Dividimos la Ecuación 7.6 por N, generando una nueva ecuación básica:

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

Demostración (Método 2) (continuación)

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

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

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1 \qquad \textbf{(7.8)}$$

$$\ldots$$

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

Las *sumas telescópicas* generan gran cantidad de términos que se cancelan.

A continuación, sumamos todos los lados izquierdos de las Ecuaciones 7.8 y lo igualamos a la suma de todos los lados derechos. Observe que el término $T(N/2)/(N/2)$ *aparece en ambos lados y por tanto se cancela. De hecho, prácticamente todos los términos aparecen en ambos lados y pueden cancelarse. Esto recibe el nombre de suma telescópica. Cuando se ha sumado y simplificado 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, de modo que todos los unos al final de esas ecuaciones se suman para producir* $\log N$*. Multiplicando por N en ambos lados, obtenemos la misma respuesta final que antes.*

Observe que si no dividiéramos por N al principio de la solución, la suma no sería telescópica. La decisión sobre la división necesaria para asegurar una suma telescópica requiere alguna experiencia y hace el método ligeramente más difícil de aplicar que la primera alternativa. Sin embargo, una vez ha encontrado el divisor correcto, la segunda alternativa tiende a generar un trabajo que cabe en una hoja de papel, dando lugar con ello a reducir los subsiguientes errores matemáticos. En contraste, el primer método es más una técnica de fuerza bruta.

Observe cuidadosamente que siempre que tenemos un algoritmo divide y vencerás que resuelve dos problemas de la mitad de tamaño con un coste lineal adicional, tendremos un tiempo de ejecución $O(N \log N)$. La siguiente sección examina lo que sucede en un entorno más general.

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

Esta sección nos da una fórmula general que permite que el número de subproblemas, el tamaño de éstos y la cantidad de trabajo adicional invertida sean todos genéricos. El resultado se puede usar tranquilamente aunque no se entienda la demostración.

En el análisis de la sección anterior veíamos que cuando se divide un problema en dos partes iguales, con un coste adicional $O(N)$, el resultado es un algoritmo con coste $O(N \log N)$. Pero, ¿qué pasa si dividimos el problema en tres partes del mismo tamaño con coste lineal adicional?, ¿o en siete partes con coste adicional cuadrático? (*Véase* el Ejercicio 7.17.) Esta sección proporciona una fórmula general para calcular el tiempo de ejecución de un algoritmo divide y vencerás. La fórmula requiere tres parámetros:

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

La demostración de la fórmula requiere cierta familiaridad con las sumas geométricas, pero no es necesario entender la demostración para utilizar la fórmula.

Teorema 7.5

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

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

Antes de demostrar el Teorema 7.5, veamos algunas aplicaciones del mismo. En el problema de la obtención de la subsecuencia de suma máxima, teníamos dos subproblemas, cada uno de ellos con la mitad de tamaño que el original, y un coste adicional lineal. Por tanto, los valores de los parámetros son $A = 2$, $B = 2$ y $k = 1$, y en consecuencia se aplica el segundo caso del Teorema 7.5, obteniendose así un valor $O(N \log N)$, que coincide con nuestros cálculos previos. Si, por ejemplo, resolvemos recursivamente tres problemas de la mitad del tamaño del problema original con un coste adicional lineal, tenemos que $A = 3$, $B = 2$ y $k = 1$, por lo que aplicaríamos la primera alternativa del teorema, obteniendo $O(N^{\log_2 3}) = O(N^{1,59})$. En este caso podemos observar que el coste adicional no contribuye al coste total del algoritmo. Cualquier sobrecarga menor que $O(N^{1,59})$ produciría el mismo tiempo de ejecución del algoritmo recursivo. Un algoritmo con tres subproblemas de la mitad de tamaño pero coste adicional cuadrático, tendría un tiempo de ejecución $O(N^2)$, pues se aplicaría el tercer caso. Ahora, la sobrecarga domina, una vez ésta excede el umbral $O(N^{1,59})$. En el umbral, la penalización es el factor logarítmico que aparece en el segundo caso. Pasemos ahora a demostrar el Teorema 7.5.

Demostración (del Teorema 7.5)

De forma similar a como hicimos en la prueba del Teorema 7.4, supondremos que N es una potencia de B. Por tanto, tendremos $N = B^M$. *Entonces* $N/B = B^{M-1}$ *y* $N^k = (B^M)^k = (B^k)^M$. *Tomaremos* $T(1) = 1$ *e ignoramos el factor constante en* $O(N^k)$. *Entonces obtenemos la siguiente ecuación básica*

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

Si dividimos en ambos lados por A^M, *obtenemos la nueva ecuación básica*

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

Si ahora escribimos esta ecuación para cada valor menor que M, obtenemos

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

Si sumamos todas las ecuaciones de la Ecuación 7.9, sucede de nuevo que la mayoría de los términos del lado izquierdo se cancelan con los correspondientes en el lado derecho, produciendo

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

de donde

$$T(N) = T(B^M) = A^M \Sigma_{i=0}^{M} \left(\frac{B^k}{A}\right)^i. \tag{7.10}$$

Si $A > B^k$, *entonces la suma es una serie geométrica con un radio menor que 1. Puesto que la suma de la correspondiente serie infinita converge a una constante, esta suma finita está también acotada por dicha constante, con lo que obtenemos la Ecuación 7.11.*

$$T(N) = O(A^M) = O(N^{\log_B A}). \tag{7.11}$$

Demostración (del Teorema 7.5) (continuación)

Cuando $A = B^k$, 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 $A^M = N^k$, obtenemos

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

Finalmente, si $A < B^k$, entonces los términos de la serie geométrica son mayores que 1 y podemos calcular la suma usando una fórmula estándar, obteniendo así

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

lo cual demuestra el tercer caso del Teorema 7.5.

7.6 Programación dinámica

La *programación dinámica* resuelve los subproblemas generados por un planteamineto recursivo, de forma no recursiva guardando los valores computados en una tabla.

Todo problema que se puede expresar recursivamente en forma matemática también se puede expresar directamente como un algoritmo recursivo. En muchos casos, hacer esto produce una mejora significativa en el rendimiento con respecto a una búsqueda exhaustiva ingenua. Cualquier fórmula matemática recursiva se podría traducir directamente a un algoritmo recursivo, pero a menudo el compilador no hace justicia al algoritmo recursivo y se obtiene un programa ineficiente. Éste es el caso del cálculo de los números de Fibonacci descritos en la Sección 7.3.4. En tales casos, podemos rescribir el algoritmo recursivo como un algoritmo no recursivo que sistemáticamente guarda las respuestas de los subproblemas resueltos en una tabla. Una técnica que hace uso de esta estrategia es la *programación dinámica*. Esta técnica se ilustra con el siguiente problema:

PROBLEMA DEL CAMBIO DE MONEDAS

Para una divisa con monedas de C_1, C_2, ..., C_N (unidades), ¿cuál es el mínimo número de monedas que se necesitan para devolver K unidades de cambio?

Los *algoritmos devoradores* toman decisiones locales óptimas en cada paso. Ésta es una forma simple de (intentar) hacer las cosas, pero no siempre funciona correctamente.

La divisa española tiene monedas de 1, 5, 10 y 25 pesetas (ignoremos las monedas de 50 pesetas, por ser poco frecuentes). Podemos juntar 63 pesetas usando dos monedas de 25 pesetas, una moneda de 10 pesetas y tres de 1 peseta, lo que hace un total de 6 monedas. Cambiar monedas en esta divisa es sencillo: usamos repetidamente la moneda con mayor valor que tengamos disponible. Se puede demostrar que para la divisa española, esto siempre minimiza el número total de monedas usadas. Éste es un ejemplo de los llamados algoritmos devoradores, en los cuales se toma una decisión en cada fase que parece ser la apropiada, sin tener en cuenta futuras consecuencias. Esta estrategia de «coge todo lo que puedas ahora» es la fuente del nombre de esta clase de algoritmos. Resulta agradable que un problema se pueda resolver usando un algoritmo devorador, ya que entonces el

algoritmo se suele acercar bastante a nuestra intuición y además el código producido resulta comprensible. Desafortunadamente los algoritmos devoradores no siempre funcionan correctamente. Si la divisa española contuviera una moneda de 21 pesetas, entonces el algoritmo devorador seguiría dando una solución de seis monedas, cuando la solución óptima usa tres (de 21 pesetas).

La pregunta se convierte entonces en cómo resolver el problema para un conjunto de monedas arbitrario. Supongamos que siempre existe una moneda de 1 unidad, con lo que nos aseguramos de que el problema tiene siempre solución. Una estrategia simple para reunir K unidades consiste en usar la recursión como sigue:

1. Si con una sola moneda ya podemos devolver el cambio solicitado, ésta (1) es la cantidad mínima de monedas.
2. En caso contrario, para cada posible valor i podemos calcular de forma independiente el número mínimo de monedas que se necesitan para reunir i y $K - i$ unidades, y elegimos el i que minimice la suma de ambos.

Se puede escribir fácilmente un algoritmo recursivo simple para cambiar monedas, pero es ineficiente.

Como ejemplo, una vez introducida la moneda de 21 unidades, veamos cómo podemos devolver 63 unidades de cambio. Claramente, una moneda no es suficiente. Podemos calcular independientemente el número de monedas requerido para devolver 1 unidad y 62 unidades de cambio (que son 1 y 4 respectivamente). Estos resultados se obtienen recursivamente, por lo que podemos asumirlos como óptimos (las 62 unidades se reúnen con dos monedas de 21 unidades y dos de 10 unidades). Obtenemos así una solución con cinco monedas. Si dividimos el problema en 2 y 61 unidades, las soluciones recursivas son respectivamente 2 y 4, dando un total de seis monedas. Continuamos probando todas las posibilidades, algunas de las cuales se muestran en la Figura 7.20. Eventualmente, llegamos a la división en 21 y 42 unidades, que se pueden obtener usando respectivamente 1 y 2 monedas, lo que hace un total de tres.

La última división probaría con 31 y 32 unidades. Podemos reunir 31 unidades con dos monedas y 32 con tres, lo que hace un total de cinco monedas. En consecuencia, el mínimo sigue siendo tres monedas.

De nuevo, cada uno de los subproblemas se está resolviendo recursivamente, lo cual genera de forma natural el algoritmo de la Figura 7.21. Si ejecutamos el algoritmo para cambiar cantidades pequeñas, funciona perfectamente, pero al igual que ocurría en el cálculo de los números de Fibonacci hay demasiado trabajo

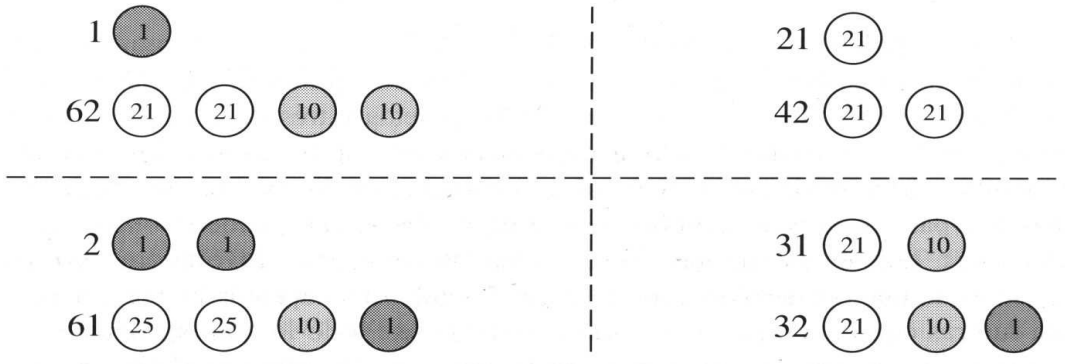

Figura 7.20 Algunos de los subproblemas resueltos recursivamente en la Figura 7.21.

```
1  // Devuelve el mínimo número de monedas para devolver un cambio
2  // Es un algoritmo recursivo simple muy ineficiente
3
4  public static int devolverCambio( int [ ] monedas, int cambio,
5                                    int monedasDistintas)
6  {
7      int minMonedas = cambio;
8
9          // Búsqueda de una moneda que reúna el cambio exacto
10     for( int i = 0; i < monedasDistintas; i++ )
11         if( monedas[i] == cambio )
12             return 1;
13
14         // No hay ninguna; se resuelve recursivamente
15     for( int j = 1; j <= cambio / 2; j++ )
16     {
17         int monedasActuales =
18           devolverCambio( monedas, j, monedasDistintas) +
19           devolverCambio( monedas, cambio - j, monedasDistintas );
20         if ( monedasActuales < minMonedas )
21              minMonedas = monedasActuales;
22     }
23
24     return minMonedas;
25 }
```

Figura 7.21 Método recursivo ineficiente para resolver el problema del cambio.

redundante. Como consecuencia, el algoritmo no terminará en una cantidad razonable de tiempo para el caso de las 63 unidades.

Un algoritmo recursivo alternativo para el problema del cambio es aún insuficiente.

Un algoritmo alternativo consiste en reducir recursivamente el tamaño del problema especificando una primera moneda. Por ejemplo, para 63 unidades, podemos dar cambio de cada una de las formas siguientes, que se muestran en la Figura 7.22:

- Una moneda de 1 unidad más el número de monedas necesarios para cambiar 62 unidades, calculado de forma recursiva.
- Una moneda de 5 unidades más el número de monedas necesarios para cambiar 58 unidades, calculado de forma recursiva.
- Una moneda de 10 unidades más el número de monedas necesarios para cambiar 53 unidades, calculado de forma recursiva.
- Una moneda de 21 unidades más el número de monedas necesarios para cambiar 42 unidades, calculado de forma recursiva.
- Una moneda de 25 unidades más el número de monedas necesarios para cambiar 38 unidades, calculado de forma recursiva.

En lugar de generar 62 problemas recursivamente, como se hizo en la Figura 7.20, ahora solamente realizamos cinco llamadas recursivas, una por cada tipo distinto de moneda. A pesar de ello, una implementación recursiva ingenua seguiría siendo ineficiente porque repetiría cálculos. Por ejemplo, en el primer caso tenemos que resolver el problema de devolver 62 unidades de cambio. En este subproblema, una de las llamadas recursivas que se hacen elige una moneda de 10 unidades y resuelve recursivamente el problema para 52 unidades. Por otro lado, en el tercer caso tenemos que resolver el problema de devolver 53 unidades. Una

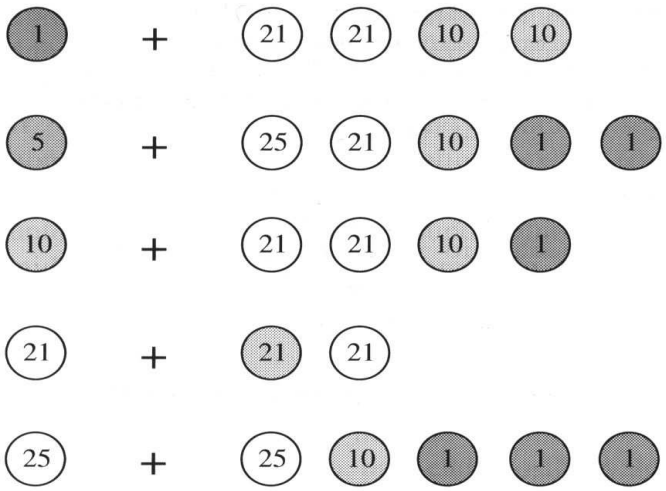

Figura 7.22 Algoritmo recursivo alternativo para el problema de la devolución de cambio.

de sus llamadas recursivas quita una unidad y resuelve recursivamente el problema para 52 unidades. Este trabajo redundante produce otra vez una gran cantidad de tiempo de ejecución. Sin embargo, si replanteamos radicalmente la forma de implementar el algoritmo, podemos hacer que el mismo sea razonablemente rápido.

El truco está en guardar las soluciones de los subproblemas en un vector. Esta técnica de programación dinámica forma la base de muchos algoritmos. Puesto que la solución a un problema grande depende solamente de las soluciones de problemas más pequeños, podemos ir calculando las soluciones óptimas correspondientes a cambiar 1 unidad, 2 unidades, 3 unidades y así sucesivamente. Esta estrategia se implementa en el programa de la Figura 7.23.

Comenzamos observando en la línea 11, que 0 unidades se pueden cambiar usando cero monedas. A continuación, a partir de la línea 13 vamos calculando el cambio óptimo para cada valor de `unidades` desde 1 hasta el valor solicitado `maxCambio`. El vector `ultimaMoneda` se usa para recordar cuál fue la última moneda usada para devolver el cambio óptimo en cada caso. Para calcular la forma óptima de devolver `unidades` unidades, probamos con cada tipo de moneda, como se indica en el bucle `for` de la línea 18. Si el valor de la moneda es mayor que la cantidad que queremos cambiar, entonces no hacemos nada. En caso contrario, en la línea 22 comprobamos si el número de monedas usado para resolver el subproblema restante más 1 (por la moneda actual) es menor que las usadas hasta el momento. Si es así, realizamos la actualización de `minMonedas` en las líneas 25 a 27. Cuando el bucle termina, en las líneas 30 y 31 se almacenan los valores obtenidos de `minMonedas` y `nuevaMoneda` en los vectores.

Al final del algoritmo, `monedasUsadas[maxCambio]` representa el mínimo número de monedas necesarias para cambiar `maxCambio` unidades. Volviendo hacia atrás a lo largo de `ultimaMoneda`, podemos averiguar las monedas necesarias para obtener la solución óptima. El tiempo de ejecución es el de los dos bucles `for` anidados, y por tanto es $O(NK)$, donde N es el número de monedas con distintos valores y K la cantidad (máxima) de cambio que queremos devolver.

```
1  // Algoritmo de programación dinámica para el problema de la devolución
2  // del cambio. Como resultado, el vector monedasUsadas se rellena con
3  // el mínimo número de monedas necesario para cambiar desde 0 hasta
4  // maxCambio unidades, y ultimaMoneda contiene una de las monedas
5  // necesarias para hacer el cambio.
6
7  public static void devolverCambio( int [ ] monedas,
8                int monedasDistintas, int maxCambio,
9                int [ ] monedasUsadas, int [ ] ultimaMoneda )
10 {
11     monedasUsadas[ 0 ] = 0; ultimaMoneda[ 0 ] = 1;
12
13     for( int unidades = 1; unidades <= maxCambio; unidades ++ )
14     {
15         int minMonedas = unidades;
16         int nuevaMoneda = 1;
17
18         for( int j = 0; j < monedasDistintas; j++ )
19         {
20             if( monedas[ j ] > unidades ) // No se puede usar la moneda j
21                 continue;
22             if( monedasUsadas[ unidades - monedas[ j ] ] + 1
23                                                     < minMonedas )
24             {
25                 minMonedas =
26                         monedasUsadas[ unidades - monedas[ j ] ] + 1;
27                 nuevaMoneda = monedas[ j ];
28             }
29         }
30         monedasUsadas[ unidades ] = minMonedas;
31         ultimaMoneda [ unidades ] = nuevaMoneda;
32     }
33 }
```

Figura 7.23 Algoritmo de programación dinámica para resolver el problema de la devolución de cambio que calcula el cambio óptimo para todas las cantidades desde 0 hasta `maxCambio` y mantiene información para construir la secuencia de monedas.

7.7 Algoritmos de vuelta atrás

Esta sección muestra la última aplicación de la recursión. Veremos, en particular, cómo escribir una rutina que haga que el computador seleccione un movimiento óptimo en el juego de las tres en raya. La clase `Mejor`, mostrada en la Figura 7.24, se usa para almacenar el movimiento óptimo devuelto por el algoritmo de selección. En la Figura 7.25 se muestra el esqueleto para una clase `TresEnRaya`. Esta clase tiene un atributo `tablero` que representa el estado actual del juego. Se especifican una serie de métodos, incluyendo rutinas para limpiar el tablero, para comprobar si una casilla está ocupada, para colocar una pieza en una casilla y para comprobar si alguno de los jugadores ha ganado. Los detalles de implementación se encuentran en el código correspondiente disponible en Internet.

```
1  final class Mejor
2  {
3      int fila;
4      int columna;
5      int valor;
6
7      public Mejor( int v )
8        { this( v, 0, 0 ); }
9      public Mejor( int v, int f, int c )
10       { valor = v; fila = f; columna = c; }
11 }
```

Figura 7.24 Clase para almacenar un movimiento evaluado.

Los *algoritmos de vuelta atrás* usan la recursión para probar sistemáticamente todas las posibilidades.

La rutina interesante es la que decide, desde cualquier posición, cuál es el mejor movimiento. Ésta es la rutina `elegirMovimiento`. La estrategia general usa un algoritmo de vuelta atrás, es decir, un algoritmo que usa recursión para probar sistemáticamente todas las posibilidades.

La base para tomar esta decisión se encuentra en la rutina `estadoActual`, que se muestra en la Figura 7.26. Dicha rutina devuelve `HUMANO_GANA`, `COMPUTADORA_GANA`, `EMPATE` o `INCIERTO`, dependiendo de la posición sobre el tablero.

En el juego de las tres en raya se usa la *estrategia minimax*, la cual considera que ambos jugadores puedan jugar de forma óptima.

La estrategia que se usa habitualmente es la *estrategia minimax*, que considera que ambos jugadores juegan de forma óptima. El valor del estado es `COMPUTADORA_GANA` si mediante una jugada óptima el computador puede forzar una victoria. Si el computador puede forzar un empate pero no una victoria, el valor del estado es `EMPATE`; si la persona puede forzar su victoria el valor del estado es `HUMANO_GANA`. Puesto que queremos que el computador gane, consideraremos `HUMANO_GANA<EMPATE<COMPUTADORA_GANA`.

Para el computador, el valor del estado es el máximo de los valores de los estados que pueden resultar de hacer un movimiento. Supongamos que un movimiento lleva a un estado ganador, dos llevan a un empate y seis llevan a uno perdedor. Entonces el estado inicial es un estado ganador porque el computador puede conseguir la victoria, sin que su rival pueda evitarlo. El correspondiente movimiento que lleva al computador a ganar es el que ésta debería hacer. Para generar el comportamiento de la persona, obramos de forma análoga pero utilizando el mínimo en lugar del máximo.

Esto sugiere un algoritmo recursivo para determinar el valor de un estado. Una vez escrito éste, llevar cuenta del mejor movimiento es simple cuestión de almacenamiento. Si el estado es final (es decir, se ha obtenido tres en raya o el tablero está lleno), el valor del estado se obtiene de forma inmediata. En caso contrario, probamos recursivamente cada uno de los movimientos posibles, calculando el valor de cada estado resultante, y elegimos el máximo valor posible. La llamada recursiva requiere entonces que el jugador humano evalúe el estado. Para él, el valor del estado es el menor de los valores de los posibles estados siguientes, puesto que el jugador humano está tratando que el computador pierda. En consecuencia, el método recursivo `elegirMovimiento`, que se muestra en la Figura 7.27, toma un parámetro `lado`, que indica a quién le toca mover.

Las líneas 12 y 13 tratan el caso base de la recursión. Si tenemos una respuesta inmediata, podemos terminar. En caso contrario, en las líneas 15 a 22 determinamos los valores adecuados de un par de variables, atendiendo a qué jugador le toca

```
1  class TresEnRaya
2  {
3      public static final int HUMANO              = 0;
4      public static final int COMPUTADORA         = 1;
5      public static final int VACIA               = 2;
6
7      public static final int HUMANO_GANA         = 0;
8      public static final int EMPATE              = 1;
9      public static final int INCIERTO            = 2;
10     public static final int COMPUTADORA_GANA    = 3;
11
12         // Constructor
13     public TresEnRaya( )
14       { limpiaTablero( ); }
15
16         // Calcula el movimiento óptimo
17     public Mejor elegirMovimiento( int lado )
18       { /* Implementación en la Figura 7.27 */ }
19
20         // Calcula el valor estático del estado actual
21     private int valorEstado( )
22       { /* Implementación en la Figura 7.26 */ }
23
24         // Hace un movimiento, comprobando su legalidad
25     public boolean movimientoJuego( int lado, int fila, int columna)
26       { /* Implementación en el código disponible en Internet */ }
27
28         // Limpia el tablero de juego
29     public void limpiaTablero( )
30       { /* Implementación en el código disponible en Internet */ }
31
32         // Devuelve true si el tablero está lleno
33     public boolean tableroLleno( )
34       { /* Implementación en el código disponible en Internet */ }
35
36         // Devuelve true si algun jugador ha ganado
37     boolean hayGanador( int lado )
38       { /* Implementación en el código disponible en Internet */ }
39
40         // Hace un movimiento, posiblemente liberando una casilla
41     private void lugar( int fila, int columna, int ficha )
42       { tablero[ fila ][ columna ] = ficha; }
43
44         // Comprueba si una casilla está vacía
45     private boolean casillaVacia( int fila, int columna )
46       { return tablero[ fila ][ columna ] = VACIA; }
47
48     private int [ ] [ ] tablero = new int[ 3 ][ 3 ];
49 }
```

Figura 7.25 Esqueleto de la clase `TresEnRaya`.

mover. El código de las líneas 28 a 38 se ejecuta una vez para cada movimiento disponible. Intentamos mover en la línea 28, evaluamos recursivamente el movimiento en la línea 29 (guardando el valor obtenido), y finalmente deshacemos el movimiento en la línea 30. Las líneas 33 y 34 comprueban si éste es el mejor movimiento hasta la fecha. Si es así, ajustamos `valor` en la línea 36 y guardamos el movimiento en la línea 37. En la línea 41, devolvemos el valor del estado en un objeto `Mejor`.

```
1       // Calcula el valor estático del estado actual
2   private int valorEstado( )
3   {
4       return hayGanador( COMPUTADORA ) ? COMPUTADORA_GANA :
5              hayGanador( HUMANO )      ? HUMANO_GANA :
6              tableroLleno( )           ? EMPATE : INCIERTO;
7   }
```

Figura 7.26 Rutina para obtener el valor de un estado.

```
1       // Calcula el movimiento óptimo
2   public Mejor elegirMovimiento( int lado )
3   {
4       int op;             // El lado opuesto
5       Mejor replica;      // Mejor réplica del oponente
6       int evalSimple;     // Resultado de una evaluación simple
7       Estado estadoActual = new Estado( tablero );
8       int mejorFila = 0;
9       int mejorColumna = 0;
10      int valor;
11
12      if( ( evalSimple = valorEstado( ) ) != INCIERTO )
13          return new Mejor( evalSimple );
14
15      if( lado == COMPUTADORA )
16      {
17          op = HUMANO; valor = HUMANO_GANA;
18      }
19      else
20      {
21          op = COMPUTADORA; valor = COMPUTADORA_GANA;
22      }
23
24      for( int fila = 0; fila < 3; fila++ )
25          for( int columna = 0; columna < 3; columna++ )
26              if( casillaVacia( fila, columna ) )
27              {
28                  lugar( fila, columna, lado );       // Intenta mover
29                  replica = elegirMovimiento( op ); // Evalúa
30                  lugar( fila, columna, VACIA ); // Deshace el movimiento
31
32                      // Actualiza si un jugador obtiene el mejor estado
33                  if( lado == COMPUTADORA && replica.valor > valor
34                     || lado == HUMANO && replica.valor < valor )
35                  {
36                      valor = replica.valor;
37                      mejorFila = fila; mejorColumna = columna;
38                  }
39              }
40
41      return new Mejor( valor, mejorFila, mejorColumna );
42  }
```

Figura 7.27 Rutina recursiva para encontrar un movimiento óptimo en las tres en raya.

La *poda alfa-beta* es una mejora del algoritmo minimax.

Aunque la Figura 7.27 resuelve de forma óptima las tres en raya, lleva a cabo excesivas búsquedas. En concreto, para elegir el primer movimiento en un tablero vacío, hace 549.946 llamadas recursivas (este número se obtiene ejecutando el programa). Usando algunos trucos algorítmicos, podemos calcular exactamente la misma información usando menos búsquedas. Uno de esos trucos se conoce como *poda alfa-beta*, y se describe en detalle en el Capítulo 10. La aplicación de la poda alfa-beta reduce el número de llamadas recursivas a sólo 18.297.

Resumen

Este capítulo ha examinado la recursión y ha demostrado que es una herramienta potente de resolución de problemas. A continuación aparecen las reglas fundamentales de la recursión, que uno nunca debería olvidar:

1. *Casos base*: se debe tener siempre al menos un caso base que pueda resolverse sin recursión.
2. *Progreso*: cualquier llamada recursiva debe progresar hacia un caso base.
3. *«Puede creerlo»:* suponga siempre que las llamadas recursivas internas funcionan correctamente.
4. *Regla de interés compuesto*: nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

La recursión tiene muchos usos, algunos de los cuales han sido discutidos en este capítulo. Se han estudiado tres importantes técnicas de diseño de algoritmos basadas en la recursión: divide y vencerás, programación dinámica y vuelta atrás.

El siguiente capítulo estudia el problema de la ordenación. El algoritmo de ordenación más rápido que se conoce es recursivo.

Elementos del juego

algoritmo de vuelta atrás Algoritmo que usa la recursión para probar sistemáticamente todas las posibilidades.

algoritmo devorador Algoritmo que toma decisiones locales óptimas en cada paso. Escribirlo suele ser simple. Sin embargo, no siempre funciona correctamente.

algoritmo divide y vencerás Tipo de algoritmo recursivo generalmente muy eficiente. La recursión es la parte «divide» y la combinación de las soluciones recursivas es el «vencerás».

caso base Instancia que se puede resolver sin hacer llamadas recursivas. Toda llamada recursiva debe hacer progresos hacia un caso base. También, en una prueba por inducción, es el caso que se puede demostrar directamente.

criptografía de clave pública Tipo de criptografía en la que cada emisor publica el código que otros pueden usar para enviar a dicho emisor mensajes codificados, pero guarda el código secreto de decodificación.

encriptación Esquema de codificación usado en la transmisión de mensajes para que no puedan ser leídos por otras personas.

estrategia minimax Estrategia usada para las tres en raya y otros juegos estratégicos que supone que ambos jugadores juegan de forma óptima.

hipótesis de inducción Hipótesis consistente en que un teorema es cierto para un caso arbitrario de partida. Se usa en las pruebas por inducción probando que a partir de dicha hipótesis se puede demostrar que el siguiente caso también es cierto.

inducción Técnica de demostración usada para demostrar teoremas que se cumplen para los enteros positivos.

inverso multiplicativo Solución $1 \leqslant X < N$ a la ecuación $AX \equiv 1 (\mathrm{mod}\ N)$.

máximo común divisor (mcd) El máximo común divisor de dos enteros es el mayor entero que los divide a ambos.

método recursivo Método que, directa o indirectamente, se hace una llamada a sí mismo.

números de Fibonacci Secuencia de números en la que el número *i*-ésimo es la suma de los dos anteriores. Los números de Fibonacci aparecen repetidamente en la literatura.

poda alfa-beta Mejora del algoritmo minimax.

programación dinámica Técnica que evita la explosión recursiva guardando respuestas en una tabla.

registro de activación Método usado para realizar las tareas de bajo nivel en muchos lenguajes de programación. Se suele usar una pila de registros de activación.

reglas de la recursión 1. *Casos base*: se debe tener siempre al menos un caso base que pueda resolverse sin llamadas recursivas. 2. *Progreso*: cualquier llamada recursiva debe progresar hacia un caso base. 3. *«Puede creerlo»:* suponga siempre que las llamadas recursivas funcionan correctamente. 4. *Regla de interés compuesto*: nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

rutina guía Rutina que comprueba la validez de los argumentos de partida y llama después a la rutina recursiva.

sistema de criptografía RSA Método popular de encriptación.

suma telescópica Desarrollos en los que se genera una gran cantidad de términos que luego se cancelan.

Errores comunes

1. El error más común cuando se usa la recursión es olvidar el caso base.
2. Asegúrese de que cada llamada recursiva hace progresos hacia un caso base, pues en caso contrario, la recursión no terminará.
3. Se debe evitar la superposición de llamadas recursivas, pues tiende a generar algoritmos exponenciales.
4. Usar la recursión en lugar de un simple bucle es un mal estilo de programación.
5. Los algoritmos recursivos se analizan usando fórmulas recursivas. No suponga alegremente que una llamada recursiva requiere un tiempo lineal.
6. Violar los derechos de propiedad (*copyright*) es también un error. RSA está patentado, aunque se permiten algunos usos libres. Consulte la bibliografía para una mayor información al respecto.

En Internet

Se proporciona todo el código que aparece en el capítulo, incluyendo un programa para el juego de las tres en raya. En el Capítulo 10 se discute un versión mejorada de este juego que usa estructuras de datos más sofisticados. Todas las rutinas están en el directorio **Chapter07,** salvo cuando se indique explícitamente otra cosa. A continuación se enumeran los nombres de los ficheros:

RecSum.java	Incluye la rutina de la Figura 7.1, con un `main` simple.
PrintInt.java	Incluye la rutina traducida en la Figura 7.4 por `ImprimeEntero`, la cual se usa para imprimir un número en cualquier base, y además una función `main`.
Factorial.java	Incluye la traducción de la rutina de la Figura 7.8 y una función `main` simple.
BinarySearch.java	Incluye la rutina traducida en la Figura 7.9 por `busquedaBinaria` y una función `main`.
Ruler.java	Incluye la rutina traducida en la Figura 7.11 por `pintaRegla`, lista para ser ejecutada. Contiene también código que ralentiza el proceso de dibujo.
FractalStar.java	Incluye la rutina traducida en la Figura 7.13 por `pintaFractal`, lista para ejecutarse. Contiene también código que ralentiza el proceso de dibujo.
Numerical.java	Incluye la versión inglesa de rutinas matemáticas de la Sección 7.4, junto con una rutina para comprobar si un número es primo, y una función `main`; todas ellas en el directorio **Supporting** (traducido por `Soporte`).
RSA.java	Incluye el algoritmo RSA.
MaxSum.java	Incluye las cuatro rutinas del problema de la obtención de la subsecuencia de suma máxima (en el directorio**Chapter05**).
MkChange.java	Incluye la rutina traducida en la Figura 7.23 por `devolverCambio` y una función `main` simple.
Best.java	Clase usada en el algoritmo de las tres en raya, aquí traducida por `Mejor`.
TicTacToe.java	Algoritmo de tres en raya básico (traducido aquí por `TresEnRaya`).
TicTacMain.java	Simple interfaz gráfica de usuario para el programa de las tres en raya.

Ejercicios

Cuestiones breves

7.1. ¿Cuáles son las cuatro reglas fundamentales de la recursión?

7.2. Modifique el programa de la Figura 7.1 de forma que si `n` es negativo, se devuelva cero. Haga el menor número de cambios.

7.3. A continuación aparecen cuatro alternativas para la línea 10 de la rutina `potencia`. Diga por qué cada una de ellas es errónea.

```
long tmp = potencia(x * x, n/2, p);
long tmp = potencia(potencia(x, 2, p), n/2, p);
long tmp = potencia(potencia(x, n/2, p), 2, p);
long tmp = potencia(x, n/2, p) * potencia(x, n/2, p) % p;
```

7.4. Muestre cómo se procesan las llamadas recursivas en el cálculo de $2^{63} \bmod 37$.

7.5. Calcule *mcd*(1995, 1492).

7.6. Roberto elige p y q iguales a 37 y 41, respectivamente. Determine valores aceptables para los parámetros restantes del algoritmo RSA.

7.7. Muestre que el algoritmo devorador para la devolución del cambio fallaría si las monedas de 5 pesetas no formaran parte de la divisa española.

Problemas teóricos

7.8. Demuestre por inducción la fórmula siguiente para F_N:

$$F_N = \frac{1}{\sqrt{5}}\left(\left(\frac{(1+\sqrt{5})}{2}\right)^N - \left(\frac{1-\sqrt{5}}{2}\right)^N\right)$$

7.9. Demuestre las siguientes identidades relativas a los números de Fibonacci:

a) $F_1 + F_2 + \ldots + F_N = F_{N+2} - 1$

b) $F_1 + F_3 + \ldots + F_{2N-1} = F_{2N}$

c) $F_0 + F_2 + \ldots + F_{2N} = F_{2N+1} - 1$

d) $F_{N-1}F_{N+1} = (-1)^N + F_N^2$

e) $F_1F_2 + F_2F_3 + \ldots + F_{2N-1}F_{2N} = F_{2N}^2$

f) $F_1F_2 + F_2F_3 + \ldots + F_{2N}F_{2N+1} = F_{2N+1}^2 - 1$

g) $F_N^2 + F_{N+1}^2 = F_{2N+1}$

7.10. Demuestre que si $A \equiv B(\bmod N)$, entonces para cualesquiera C, D y P, se cumple:

a) $A + C \equiv B + C(\bmod N)$

b) $AD \equiv BD(\bmod N)$

c) $A^P \equiv B^P(\bmod N)$

7.11. Demuestre que si $A \geqslant B$, entonces $A \bmod B < A/2$. (*Pista*: considere por separado los casos $B \leqslant A/2$ y $B > A/2$.) ¿Cómo nos ayuda esto a demostrar que el tiempo de ejecución de `mcd` es logarítmico?

7.12. Demuestre por inducción la fórmula que define el número de llamadas del método recursivo `fib` de la Sección 7.3.4.

7.13. Demuestre por inducción que si $A > B \geqslant 0$ y la llamada a `mcd(a, b)` lleva a cabo $k \geqslant 1$ llamadas recursivas, entonces $A \geqslant F_{k+2}$ y $B \geqslant F_{k+1}$.

7.14. Demuestre por inducción que en el algoritmo extendido del *mcd*, $|X| < B$ y $|Y| < A$.

7.15. Escriba un algoritmo alternativo para el cálculo del *mcd* basado en las siguientes observaciones (supuesto que $A > B$):

a) $mcd(A, B) = 2\ mcd(A/2, B/2)$ si A y B son pares.

b) $mcd(A, B) = mcd(A/2, B)$ si A es par y B impar.

c) $mcd(A, B) = mcd(A, B/2)$ si A es impar y B es par.
d) $mcd(A, B) = mcd((A + B)/2, (A - B)/2)$ si A y B son impares.

7.16. Resuelva la siguiente ecuación, suponiendo que $A \geq 1$, $B > 1$ y $P \geq 0$.

$$T(N) = AT(N/B) + O(N^k \log^P N)$$

7.17. El algoritmo de Strassen para multiplicar matrices lleva a cabo la multiplicación de dos matrices $N \times N$ a través de siete llamadas recursivas que calculan el producto de dos matrices $N/2 \times N/2$. El coste adicional es cuadrático. ¿Cuál es el tiempo de ejecución del algoritmo de Strassen?

Problemas prácticos

7.18. El método `imprimeEntero` de la Figura 7.4 trata de forma incorrecta el caso en el que $N =$ `Integer.MIN_VALUE`. Explique por qué y proporcione una solución.

7.19. Escriba un método recursivo que devuelva el número de unos en la representación binaria de N. Use el hecho de que es igual al número de unos en la representación binaria de $N/2$, más 1 si N es impar.

7.20. Implemente de forma recursiva la búsqueda binaria con una comparación por nivel.

7.21. El algoritmo de obtención de la subsecuencia de suma máxima de la Figura 7.18 no devuelve la subsecuencia en cuestión, ni ningún otro valor a partir del cual podamos obtenerla fácilmente. Modifíquelo de forma que devuelva en un único objeto el valor de la subsecuencia máxima y los índices de la subsecuencia (de forma similar a como se hizo en `elegirMovimiento`). Después haga que la rutina guía determine los atributos estáticos, igual que en la Sección 5.3.

7.22. Proporcione un algoritmo para el problema del cambio que calcule el número de formas distintas de devolver K unidades de cambio.

7.23. El problema del *subconjunto suma* es el siguiente: dado un conjunto de N enteros, $A_1, A_2, ..., A_N$ y un entero K, determinar si hay un subconjunto de enteros que sumen exactamente K. Proporcione un algoritmo $O(NK)$ para resolver el problema.

7.24. Proporcione un algoritmo $O(2^N)$ para el problema del subconjunto suma descrito en el Ejercicio 7.23. *Pista*: use recursión.

7.25. Escriba sendas rutinas con las siguientes declaraciones:

```
public void permutacion( String str );
private void permutacion( char [ ] str, int inf, int sup );
```

La primera rutina es una guía que llama a la segunda e imprime todas las permutaciones de los caracteres del `String str`. Si `str` fuera `"abc"`, entonces las cadenas de salida serían `abc`, `acb`, `bca`, `cab` y `cba`. Use recursión en la segunda rutina.

7.26. Escriba un programa que calcule el máximo valor de N que puede manejar en su máquina el programa de la Figura 7.1. *Pista*: use la idea de la búsqueda binaria para determinar N; el mecanismo para decidir si una llamada a `s` tiene éxito consiste en comprobar si se ha lanzado alguna excepción de la clase `Throwable`.

7.27. Escriba un programa que dibuje la Figura 7.7.

7.28. Explique lo que sucede si en la Figura 7.13 dibujamos el cuadrado central antes de hacer las llamadas recursivas.

Prácticas de programación

7.29. Los coeficientes binomiales $C(N, k)$ se pueden definir recursivamente en la siguiente forma:

$$C(N, 0) = 1,\ C(N, N) = 1$$

y para

$$0 < k < N,\ C(N, k) = C(N - 1, k) + C(N - 1, k - 1)$$

Escriba un método para calcular los coeficientes binomiales y proporcione un análisis de su tiempo de ejecución. Escriba el método usando:

a) Recursión.
b) Programación dinámica.

7.30. Implemente el sistema de criptografía RSA usando una clase de librería `BigDecimal`.

7.31. Mejore la implementación de las tres en raya haciendo las rutinas de soporte más eficientes y la interfaz gráfica de usuario más elaborada que la que se proporciona en Internet.

7.32. Sea A una secuencia de N números distintos ordenados A_1, A_2, ..., A_N con $A_1 = 0$. Sea B una secuencia de $N(N - 1)/2$ números, definidos por $B_{i,j} = A_i - A_j$ $(i < j)$. Sea D la secuencia resultante de la ordenación de B. Tanto B como D pueden contener elementos repetidos. Por ejemplo, podríamos tener $A = 0, 1, 5, 8$, y entonces $D = 1, 3, 4, 5, 7, 8$. Haga lo siguiente:

a) Escriba un programa que construya D a partir de A. Esta parte es fácil.
b) Escriba un programa que construya una secuencia A correspondiente a D. Observe que A no es única. Use un algoritmo de vuelta atrás.

7.33. Considere una cuadrícula $N \times N$ en la que algunas casillas están ocupadas por círculos negros. Dos casillas ocupadas pertenecen al mismo grupo si comparten un lado común. En la Figura 7.28, hay un grupo de cuatro casillas ocupadas, tres grupos de dos casillas ocupadas y dos casillas individuales ocupadas. Supongamos que la cuadrícula se representa mediante un vector bidimensional. Escriba un programa que:

a) Calcule el tamaño del grupo al que pertenece una casilla ocupada.
b) Calcule el número de grupos distintos.
c) Liste todos los grupos.

7.34. Escriba un programa que expanda las directivas `#include` de los ficheros fuente escritos en C o C++. Para ello reemplace líneas de la forma

```
#include "nombrefichero"
```

por el contenido de `nombrefichero`.

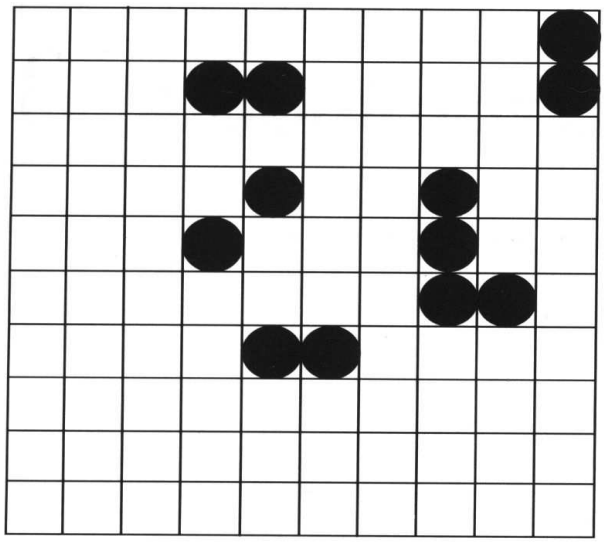

Figura 7.28 Cuadrícula del Ejercicio 7.33.

Bibliografía

Una gran parte de este capítulo está basada en [4]. En [1] se puede encontrar una descripción del algoritmo RSA junto con una prueba de corrección, y además un capítulo dedicado a la programación dinámica. El algoritmo RSA está patentado, y su uso comercial requiere el pago de una licencia; pero existe una implementación llamada *PGP (pretty good privacy)* que está disponible y cuyo uso no comercial está permitido sin necesidad de pago. En [3] se pueden encontrar más detalles sobre el particular. Los ejemplos de elaboración de los dibujos se han adaptado de [2].

1. T. H. Cormen, C. E. Leiserson, y R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Mass. (1990).

2. R. Sedgewick, *Algorithms in C++*, Addison-Wesley, Reading, Mass. (1992).

3. W. Stallings, *Protect Your Privacy: A Guide for PGP Users*, Prentice-Hall, Englewood Cliffs, NJ (1995).

4. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).