Capítulo 7

Recursión

A un método que esté parcialmente definido en términos de sí mismo se le denomina *recursivo*. Al igual que otros muchos lenguajes, Java soporta los métodos recursivos. La recursión, que es el uso de métodos recursivos, es una potente herramienta de programación que en muchos casos puede proporcionar algoritmos que son, a la vez, cortos y eficientes. En este capítulo vamos a explorar cómo funciona la recursión, proporcionando así ideas acerca de sus variaciones, limitaciones y uso. Comenzaremos nuestras explicaciones sobre la recursión examinando el principio matemático en el que se basa esta técnica: la *inducción matemática*. Después proporcionaremos ejemplos de métodos recursivos simples y demostraremos que esos métodos generan las respuestas correctas.

En este capítulo, vamos a ver

- Las cuatro reglas básicas de la recursión.
- Aplicaciones numéricas de la recursión, que conducen a la implementación de un algoritmo de cifrado.
- Una técnica general denominada *divide y vencerás*.
- Una técnica general denominada *programación dinámica*, que es similar a la recursión, pero que utiliza tablas en lugar de emplear llamadas a métodos recursivos.
- Una técnica general denominada *retroceso*, que equivale a una cuidadosa búsqueda exhaustiva.

7.1 ¿Qué es la recursión?

Un *método recursivo* es un método que hace, directa o indirectamente, una llamada a sí mismo. Esta acción puede parecer similar a los razonamientos circulares: ¿cómo puede un método F resolver un problema llamándose a sí mismo? La clave está en que el método F se llama a sí mismo dentro de un contexto diferente, que generalmente es más simple que el anterior. He aquí algunos ejemplos.

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

- Los archivos de una computadora suelen estar almacenados en directorios. Los usuarios pueden crear subdirectorios que almacenen más archivos y directorios. Suponga que deseamos examinar cada archivo de un directorio D incluyendo todos los archivos de todos los

subdirectorios (y de sus sub-subdirectorios, etc.). Podemos hacerlo examinando recursivamente cada archivo de cada subdirectorio y luego examinando todos los archivos del directorio *D* (como se explica en el Capítulo 18).

- Suponga que tenemos un diccionario de gran tamaño. Las palabras de los diccionarios se definen utilizando otras palabras. Cuando buscamos el significado de un término puede que, en ocasiones, no entendamos la definición y que tengamos que buscar el significado de algunas de las palabras que esa definición contiene. De forma similar, podríamos no entender algunas de las definiciones de esas otras palabras, lo que nos llevaría a tener que continuar con nuestra búsqueda durante un cierto tiempo. Como el diccionario es finito, terminaremos llegando a un punto en el que comprenderemos todas las palabras de una cierta definición (y por tanto entenderemos la propia definición gracias a las otras definiciones), o en el que encontremos que las definiciones son circulares y estamos en un callejón sin salida o en el que comprobemos que alguna palabra que necesitamos comprender no está definida en el diccionario. Nuestra estrategia recursiva para comprender las palabras es la siguiente: si sabemos el significado de una palabra, habremos terminado; en caso contrario, la buscamos en el diccionario. Si comprendemos todas las palabras de la definición habremos terminado. En caso contrario, intentamos averiguar qué significa la definición buscando recursivamente aquellas palabras que no conozcamos. Este procedimiento terminará siempre si el diccionario está bien definido, aunque podríamos entrar en un bucle infinito si una cierta palabra estuviera definida de forma circular.
- Los lenguajes informáticos suelen definirse frecuentemente de manera recursiva. Por ejemplo, una expresión aritmética es un objeto o una expresión entre paréntesis, dos expresiones sumadas, etc.

La recursión es una potente herramienta de resolución de problemas. La manera más fácil de expresar muchos algoritmos es mediante una formulación recursiva. Además, las soluciones más eficientes a muchos problemas están basadas en esta formulación recursiva natural. Pero debemos tener cuidado de no crear una lógica circular que termine haciéndonos entrar en un bucle infinito.

En este capítulo, vamos a explicar las condiciones generales que deben satisfacer los algoritmos recursivos y vamos a proporcionar varios ejemplos prácticos. Veremos que en ocasiones existen algoritmos que se expresan naturalmente de forma recursiva, pero que deben ser reescritos sin utilizar la recursión.

7.2 Fundamentos: demostraciones por inducción matemática

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

En esta sección vamos a hablar del concepto de demostración por *inducción* matemática. (A lo largo del capítulo omitiremos la palabra matemática a la hora de describir esta técnica.) La *inducción* suele emplearse para demostrar teoremas que se cumplen para enteros positivos. Vamos a comenzar demostrando un teorema simple, el Teorema 7.1. Este teorema concreto puede demostrarse fácilmente mediante otros métodos, pero a menudo la demostración por inducción resulta ser el mecanismo de demostración más simple.

Teorema 7.1

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

Obviamente, el teorema es cierto para $N = 1$ porque tanto el lado izquierdo como el derecho de la igualdad tienen el valor 1. Una comprobación adicional muestra que también es cierto para $2 \leq N \leq 10$. Sin embargo, el hecho de que el teorema se cumpla para todos los valores N que se pueden comprobar fácilmente a mano, no implica que se cumpla para todo posible valor de N. Considere, por ejemplo, los números de la forma $2^{2^k} + 1$. Los primeros cinco números (correspondientes a $0 \leq k \leq 4$) son 3, 5, 17, 257 y 65.537. Todos estos números son primos. De hecho, hubo una época en la que los matemáticos plantearon la conjetura de que todos los números de esta forma son primos. Sin embargo, resulta que no es cierto. Podemos demostrarlo fácilmente, comprobando mediante una computadora que $2^{2^5} + 1 = 641 \times 6.700.417$. De hecho, no se conoce ningún otro número primo de la forma $2^{2^k} + 1$, aparte de los ya citados.

Una demostración por inducción se lleva a cabo en dos etapas. En primer lugar, como acabamos de hacer, demostramos que el teorema es cierto para los valores más pequeños, luego demostramos que si el teorema es cierto para los primeros casos, puede ampliarse para incluir el caso siguiente. Por ejemplo, demostramos que si un teorema es cierto para todo $1 \leq N \leq k$, entonces también debe ser cierto para $1 \leq N \leq k + 1$. Una vez que hayamos demostrado cómo ampliar el rango de casos para los que el teorema se cumple, habremos demostrado que se cumple para todos los casos. La razón es, obviamente, que podemos ampliar indefinidamente el rango de casos para los que el teorema se cumple. Vamos a utilizar esta técnica para demostrar el Teorema 7.1.

Una demostración por inducción demuestra que el teorema se cumple para algunos casos simples y luego demuestra cómo ampliar indefinidamente el rango de casos para los que el teorema es cierto.

Demostración del Teorema 7.1

Evidentemente, el teorema es cierto para $N = 1$. Suponga que el teorema es cierto para todo $1 \leq N \leq k$. Entonces

$$\sum_{i=1}^{k+1} i = (k+1) + \sum_{i=1}^{k} i \tag{7.1}$$

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

$$\sum_{i=1}^{k+1} i = (k+1) + k(k+1)/2) \tag{7.2}$$

Una manipulación algebraica del lado derecho de la Ecuación 7.2 nos da

$$\sum_{i=1}^{k+1} i = (k+1)(k+2)/2$$

Este resultado confirma el teorema para el caso $k + 1$. Por tanto, por inducción, el teorema es cierto para todos los enteros $N \geq 1$.

¿Por qué esto constituye una demostración? En primer lugar, el teorema es cierto para $N = 1$, lo que se denomina la *base*. Podemos contemplar este hecho como si fuera la base de nuestra creencia de que el teorema es cierto con carácter general. En una demostración por inducción, la base es el caso fácil que puede demostrarse a mano. Una vez que hayamos establecido la base, utilizamos la *hipótesis inductiva* para asumir que el teorema es cierto para un valor k

En una demostración por inducción, la *base* es el caso sencillo que puede ser demostrado a mano.

La *hipótesis inductiva* supone que el teorema es cierto para un caso arbitrario y que, bajo esa suposición, también es cierto para el caso siguiente.

arbitrario y luego, con esa suposición, demostramos que si el teorema es cierto para k, entonces es cierto para $k + 1$. En nuestro caso, sabemos que el teorema es cierto para la base $N = 1$, así que también es cierto para $N = 2$. Puesto que es cierto para $N = 2$, tiene que ser cierto para $N = 3$. Y como es cierto para $N = 3$, tiene que serlo para $N = 4$. Ampliando esta manera de razonar, sabemos que el teorema es cierto para todo entero positivo a partir de $N = 1$.

Apliquemos la demostración por inducción a un segundo problema, no tan simple como el primero. En primer lugar, examinemos 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 de los primeros N cuadrados, con signos alternativos. El valor de esta secuencia será 1, 3, 6, 10 y 15. Por tanto, en general, la suma parece ser igual a la suma de los N primeros enteros, la cual es igual, como ya sabemos por el Teorema 7.1, a $N(N + 1)/2$. El Teorema 7.2 demuestra este resultado.

Teorema 7.2 La suma $\sum_{i=N}^{1}(-1)^{N-i}i^2 = N^2 - (N-1)^2 + (N-2)^2 - \cdots$ es $N(N+1)/2$.

Demostración

La demostración es por inducción.

Base: evidentemente, el teorema es cierto para $N = 1$.

Hipótesis inductiva: en primer lugar, suponemos que el teorema es cierto para k:

$$\sum_{i=k}^{1}(-1)^{k-i}i^2 = \frac{k(k+1)}{2}$$

A continuación deberemos demostrar que es cierto para $k + 1$,

$$\sum_{i=k+1}^{1}(-1)^{k+1-i}i^2 = \frac{(k+1)(k+2)}{2}$$

Escribimos

$$\sum_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - k^2 + (k-1)^2 - \cdots \qquad (7.3)$$

Si volvemos a escribir el lado derecho de la ecuación de la Ecuación 7.3, obtenemos

$$\sum_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - (k^2 - (k-1)^2 + \cdots)$$

y sustituyendo obtenemos

$$\sum_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - \left(\sum_{i=k}^{1}(-1)^{k-i}i^2\right) \qquad (7.4)$$

Si aplicamos la hipótesis inductiva, entonces podemos sustituir el sumatorio del lado derecho de la Ecuación 7.4, obteniendo

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

Una simple manipulación algebraica en el lado derecho de la Ecuación 7.5 nos da

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

lo que demuestra el teorema para $N = k + 1$. Por tanto, por inducción, el teorema es cierto para todo $N \geq 1$.

7.3 Recursión básica

Un método recursivo se define en términos de una instancia más pequeña de sí mismo. Debe existir algún caso base que pueda calcularse sin necesidad de usar la recursión.

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

En ocasiones, hay funciones matemáticas que se definen de forma recursiva. Por ejemplo, sea $S(N)$ la suma de los N primeros enteros. Entonces, $S(1) = 1$ y podemos escribir $S(N) = S(N - 1) + N$. Aquí hemos definido la función S en términos de una instancia más pequeña de sí misma. La definición recursiva de $S(N)$ es prácticamente idéntica a la forma explícita $S(N) = N(N+ 1) / 2$, con la única excepción de que la definición recursiva solo está definida para enteros positivos y se puede calcular de una forma menos directa.

En ocasiones, escribir una fórmula recursivamente es más fácil que escribirla en forma explícita. La Figura 7.1 muestra una implementación directa de la función recursiva. Si $N = 1$, tendremos la base, para la cual sabemos que $S(1) = 1$. Tenemos este caso en las líneas 4 y 5. En caso contrario, aplicamos la definición recursiva $S(N) = S(N - 1) + N$ precisamente en la línea 7. Es complicado imaginar que se pudiera implementar el método recursivo de una forma más simple que esta, por lo que la cuestión natural que se plantea es: ¿funciona realmente el método?

La respuesta, salvo por lo que en breve comentaremos, es que esta rutina funciona perfectamente. Examinemos cómo se evaluaría la llamada a `s(4)`. Cuando se hace la llamada a `s(4)`, la comprobación de la línea 4 falla. A continuación ejecutamos la línea 7, en la que se evalúa `s(3)`. Como sucede con cualquier otro método, esta evaluación requiere una llamada a `s`. En dicha llamada, llegamos a la línea 4, donde la comprobación falla; por tanto, pasamos a la línea 7. En ese punto, llamamos a `s(2)`. De nuevo, llamamos a `s` y ahora `n` será 2. La comprobación de la línea 4 sigue fallando, así que invocamos `s(1)` en la línea 7. Ahora tenemos que `n` es igual a 1, por lo que `s(1)` devolverá 1. En este punto, `s(2)` puede continuar, sumando el valor de retorno de `s(1)` a 2; por tanto, `s(2)` devuelve 3. Ahora `s(3)` continuará su ejecución, sumando el valor 3 devuelto por `s(2)` a `n`, lo que es 3; por tanto `s(3)` devolverá 6. Este resultado permite completar la llamada a `s(4)`, que termina por devolver el valor 10.

Observe que, aunque `s` parece estar llamándose a sí mismo, en realidad está llamando a un clon de sí mismo. Ese clon es simplemente otro método con parámetros distintos. En cualquier instante determinado solo habrá un clon activo; el resto de los clones tendrán su ejecución pendiente. Es misión de la computadora, no nuestra, encargarse de gestionar todas esas llamadas. Si esas tareas

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

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

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

de gestión implicaran una carga excesiva para la computadora, entonces ya sí que tendríamos que preocuparnos. Hablaremos de estos detalles más adelante en el capítulo.

Un *caso base* es una instancia que se puede resolver sin necesidad de recursión. Toda llamada recursiva debe progresar hacia el caso base, si queremos que la ejecución termine en algún momento. Con esto podemos plantear nuestras dos primeras (de un total de cuatro) *reglas de recursión* fundamentales.

1. *Caso base*: siempre tiene que haber al menos un caso que se pueda resolver sin utilizar recursión.
2. *Progresión*: toda llamada recursiva debe progresar hacia un caso base.

Nuestra rutina de evaluación recursiva presenta, de todos modos, algunos problemas. Uno de ellos es la llamada a `s(0)`, para la que el método no se comporta adecuadamente.[1] Este comportamiento es natural porque la definición recursiva de $S(N)$ no permite que $N < 1$. Podemos resolver este problema ampliando la definición de $S(N)$ para incluir $N = 0$. Puesto que no hay ningún número que sumar en este caso, un valor natural para $S(0)$ sería 0. Este valor tiene sentido, porque la definición recursiva puede seguir aplicándose a $S(1)$, ya que $S(0) + 1$ es 1. Para implementar este cambio, simplemente sustituimos 1 por 0 en las líneas 4 y 5. Los valores negativos de N también hacen que se produzcan errores, pero este problema se puede resolver de forma similar y se deja como tarea para el lector en el Ejercicio 7.6.

Un segundo problema es que, si el parámetro `n` es grande, pero no tan grande que la respuesta no pueda caber en un `int`, el programa puede fallar o quedarse colgado. Nuestro sistema informático, por ejemplo, no puede manejar los casos de $N \geq 8.882$. La razón es que, como ya hemos comentado, la implementación de la recursión requiere que el sistema lleve la cuenta de las llamadas recursivas pendientes, y para cadenas de recursión suficientemente grandes, la computadora simplemente se queda sin memoria. Explicaremos esta condición con más detalle posteriormente en el capítulo. Esta rutina, asimismo, requiere algo más de tiempo de ejecución que la utilización de un bucle equivalente, debido a que ese control de las llamadas recursivas pendientes también requiere un cierto tiempo.

No hace falta decir que este ejemplo concreto no ilustra el mejor uso de la recursión, ya que se trata de un problema que se puede resolver muy fácilmente sin utilizar esta técnica. La mayoría de las buenas aplicaciones de la recursión no agotan la memoria de la computadora y solo consumen un tiempo ligeramente mayor que las implementaciones no recursivas. A cambio de ese ligero incremento en el tiempo de ejecución, la recursión casi siempre permite obtener un código más compacto.

7.3.1 Impresión de números en cualquier base

Un buen ejemplo de cómo la recursión simplifica la codificación de rutinas es la impresión de números. Suponga que queremos imprimir un número no negativo N en forma decimal y que no tenemos disponible una función de impresión de números. Sin embargo, imagine que sí que

[1] Se realiza una llamada a `s(-1)` y el programa termina fallando debido a que se alcanza un punto en el que habrá demasiadas llamadas recursivas pendientes. La llamadas recursivas no irán progresando hacia un caso base.

podemos imprimir un dígito cada vez. Considere, por ejemplo, en esas circunstancias, cómo podríamos imprimir el número 1369. En primer lugar, tendríamos que imprimir 1, luego 3, después 6 y luego 9. El problema es que obtener el primer dígito es algo engorroso: dado un número n, necesitamos un bucle para determinar el primer dígito de n. Por el contrario, el último dígito siempre está disponible de forma inmediata mediante n%10 (que es n para n menor que 10).

La recursión proporciona una solución elegante. Para imprimir 1369, imprimimos 136, seguido del último dígito, 9. Como ya hemos mencionado, imprimir el último dígito empleando el operador % es sencillo. Imprimir el número que resulta de eliminar el último dígito también es sencillo, porque es el mismo problema que imprimir n/10. Por tanto, podemos realizar esa tarea mediante una llamada recursiva.

El código mostrado en la Figura 7.2 implementa esta rutina de impresión. Si n es menor de 10, la línea 6 no se ejecuta y solo se imprime el dígito n%10; en caso contrario, se imprimen recursivamente todos los dígitos menos el último y después se imprime el último dígito.

Observe que tenemos un caso base (que n sea un entero de un solo dígito) y como el problema recursivo tiene un dígito menos, todas las llamadas recursivas irán progresando hacia el caso base. Por tanto, habremos satisfecho las dos primeras reglas fundamentales de la recursión.

Para que nuestra rutina de impresión sea útil, podemos ampliarla para que nos permita imprimir en cualquier base comprendida entre 2 y 16.[2] Esta modificación se muestra en la Figura 7.3.

```
1    // Imprimir n en base 10, de forma recursiva.
2    // Precondición: n >= 0.
3    public static void printDecimal( long n )
4    {
5       if( n >= 10 )
6          printDecimal( n / 10 );
7       System.out.print( (char) ('0' + ( n % 10 ) ) );
8    }
```

Figura 7.2 Una rutina recursiva para imprimir *N* en forma decimal.

```
1    private static final String DIGIT_TABLE = "0123456789abcdef";
2
3    // Imprimir n en cualquier base de forma recursiva.
4    // Precondición: n >= 0, base es válida.
5    public static void printInt( long n, int base )
6    {
7       if( n >= base )
8          printInt( n / base, base );
9       System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
10   }
```

Figura 7.3 Una rutina recursiva para imprimir *N* en cualquier base.

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

Hemos introducido una constante `String` para hacer que la impresión de `a` a `f` sea más fácil. Cada dígito se imprime ahora indexando la cadena de caracteres `DIGIT_TABLE`. La rutina `printInt` no es robusta. Si `base` es mayor que 16, el índice para acceder a `DIGIT_TABLE` podría salirse del rango. Si `base` es igual a 0, se producirá un error aritmético cuando se intente hacer una división por 0 en la línea 8.

El fallo a la hora de progresar hacia el caso base implica que el programa no funciona.

El error más interesante ocurre cuando `base` es 1. Entonces la llamada recursiva de la línea 8 no consigue progresar hacia el caso base, porque los dos parámetros de la llamada recursiva son idénticos a los de la llamada original. Por tanto, el sistema realizará llamadas recursivas hasta que termine quedándose sin espacio de memoria (y termine de forma poco grácil).

Una *rutina de preparación* comprueba la validez de la primera llamada y luego invoca la rutina recursiva.

Podemos hacer la rutina más robusta añadiendo una comprobación explícita para `base`. El problema con esta estrategia es que la comprobación se realizaría en cada una de las llamadas recursivas a `printInt`, no solo durante la primera llamada. Pero si `base` era válida en la primera llamada, volver a comprobarla es absurdo, porque no va a cambiar durante el curso de la recursión y por tanto seguirá siendo válida. Una forma de evitar esta ineficiencia consiste en programar una rutina de preparación. La *rutina de preparación* comprueba la validez de `base` y luego invoca a la rutina recursiva, como se muestra en la Figura 7.4. El uso de rutinas de preparación para los programas recursivos es una técnica bastante común.

7.3.2 Por qué funciona

En el Teorema 7.3 vamos a demostrar, de forma en cierto modo rigurosa, que el algoritmo `printDecimal` funciona. Nuestro objetivo es verificar que el algoritmo es correcto, por lo que la demostración se basa en la suposición de que no hemos cometido ningún error sintáctico.

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

La demostración del Teorema 7.3 ilustra un principio importante. A la hora de diseñar un algoritmo recursivo, siempre podemos asumir que las llamadas recursivas funcionan (si van progresando hacia el caso base) porque, cuando se desarrolla una demostración, esta suposición se emplea como hipótesis inductiva.

A primera vista, dicha suposición parece extraña. Sin embargo, recuerde que siempre asumimos que las llamadas a métodos funcionan, y por tanto la suposición de que la llamada recursiva funciona no es en realidad distinta. Al igual que con cualquier otro método, una rutina recursiva necesita combinar soluciones obtenidas a partir de llamadas a otros métodos, para obtener una solución final. No obstante, nada impide que los otros métodos puedan incluir instancias más simples del método original.

Teorema 7.3 El algoritmo `printDecimal` mostrado en la Figura 7.2 imprime correctamente `n` en base 10.

Demostración

Sea k el número de dígitos en `n`. La demostración es por inducción sobre k.

Base: si $k = 1$, entonces no se realiza ninguna llamada recursiva y la línea 7 imprime correctamente el único dígito de `n`.

Continúa

Demostración (cont.)

Hipótesis inductiva: suponga que `printDecimal` funciona correctamente para todos los enteros con un número de dígitos $k \geq 1$. Demostraremos que esta suposición implica la corrección para cualquier entero `n` de $k + 1$ dígitos. Puesto que $k \geq 1$, la instrucción `if` de la línea 5 se satisface para un entero de $k + 1$ dígitos. Por la hipótesis inductiva, la llamada recursiva de la línea 6 imprime los primeros k dígitos de `n`. Entonces, la línea 7 imprime el dígito final. Por tanto, si puede imprimirse cualquier entero de k dígitos, entonces también podrá imprimirse un entero de $k + 1$ dígitos. Por inducción, concluimos que `printDecimal` funciona para todo valor de k y por tanto para todo valor `n`.

```
1  public final class PrintInt
2  {
3     private static final String DIGIT_TABLE = "0123456789abcdef";
4     private static final int MAX_BASE = DIGIT_TABLE.length( );
5
6     // Imprimir n en cualquier base, de forma recursiva
7     // Precondición: n >= 0, 2 <= base <= MAX_BASE
8     private static void printIntRec( long n, int base )
9     {
10       if( n >= base )
11          printIntRec( n / base, base );
12       System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
13    }
14
15    // Rutina de preparación
16    public static void printInt( long n, int base )
17    {
18       if( base <= 1 || base > MAX_BASE )
19          System.err.println( "Cannot print in base " + base );
20       else
21       {
22          if( n < 0 )
23          {
24             n = -n;
25             System.out.print( "-" );
26          }
27          printIntRec( n, base );
28       }
29    }
30 }
```

Figura 7.4 Un programa robusto de impresión de números.

Esta observación nos lleva a la tercera regla fundamental de la recursión.

3. *"Es necesario creer"*: asuma siempre que la llamada recursiva funciona.

La tercera regla fundamental de la recursión: asuma siempre que la llamada recursiva funciona. Utilice esta regla para diseñar sus algoritmos.

La regla 3 nos dice que a la hora de diseñar un método recursivo, no tenemos por qué intentar trazar la posiblemente larga serie de llamadas recursivas. Como hemos visto anteriormente, esta tarea puede ser muy complicada y tiende a dificultar el diseño y la verificación. Cualquier buen uso de la recursión hace que esa tarea de trazado sea casi imposible de comprender. Intuitivamente, lo que estamos haciendo es dejar que la computadora se encargue de realizar toda la gestión que, en caso de que nos encargáramos nosotros de hacerla, daría como resultado un código mucho más largo.

Este principio es tan importante que debemos formularlo de nuevo: *asuma siempre que la llamada recursiva funciona.*

7.3.3 Cómo funciona

Recuerde que la implementación de la recursión requiere una serie de tareas adicionales de gestión por parte de la computadora. Dicho de otra forma, la implementación de cualquier método requiere una cierta gestión, y las llamadas recursivas no tienen nada de especial a este respecto (salvo porque pueden agotar las capacidades de gestión de la computadora al invocarse una rutina recursiva a sí misma demasiadas veces).

Para ver cómo puede una computadora gestionar la recursión o de modo más general, cualquier secuencia de llamadas a métodos, considere la manera en que una persona cualquiera podría gestionar un día muy atareado. Imagine que estamos editando un archivo en nuestra computadora y suena el teléfono. Al sonar el teléfono de nuestro domicilio tenemos que dejar de editar el archivo para atender la llamada. Puede que queramos escribir en un papel lo que estábamos haciendo en el archivo por si acaso la llamada telefónica dura mucho tiempo y luego no nos acordamos. Ahora imagine que, mientras que se encuentra hablando por teléfono con su esposa, le suena el teléfono móvil. Entonces deja a su esposa en espera, depositando el teléfono sobre la mesa. Puede anotar en un papel que ha dejado el teléfono descolgado sobre la mesa. Mientras está hablando por el teléfono móvil, alguien llama a la puerta. Entonces puede decirle a su interlocutor que espere mientras va a abrir la puerta para ver quién es. Así que deja el teléfono móvil sobre otro mueble, escribe en otro papel que ha dejado el móvil en un cierto lugar con una llamada en espera y abre la puerta. Llegados a este punto, habrá escrito tres notas para sí mismo, siendo la correspondiente al teléfono móvil la más reciente. Al abrir la puerta, se dispara la alarma antirrobo, porque se le ha olvidado desactivarla, así que tiene que decirle a la persona que está en la puerta que se espere. Entonces escribe otra nota para sí mismo, recordándole que hay alguien en la puerta, mientras desactiva la alarma antirrobo. Aunque está abrumado, ahora puede terminar de gestionar todas las tareas que había iniciado, en orden inverso: tratando primero con la persona de la puerta, terminado luego la conversación por el teléfono móvil, concluyendo luego la conversación con su esposa a través del teléfono fijo y finalizando después con la edición del archivo. Simplemente tiene que retroceder a través de la pila de anotaciones que ha ido haciendo para sí mismo. Observe la palabra que hemos utilizado: lo que habrá estado haciendo durante este tiempo es gestionar una "pila" de anotaciones.

Java, al igual que otros lenguajes como C++, implementa los métodos utilizando una pila interna de registros de activación. Cada *registro de activación* contiene información relevante

acerca del método, incluyendo, por ejemplo, los valores de sus parámetros y de sus variables locales. El contenido concreto de un registro de activación dependerá del sistema con el que estemos trabajando.

La gestión de invocaciones de métodos en un lenguaje procedimental y orientado a objetos se lleva a cabo utilizando una pila de *registros de activación*. La recursión es un subproducto natural de este sistema de trabajo.

La pila de registros de activación se utiliza porque los métodos vuelven en orden inverso a su invocación. Recuerde que las pilas son muy útiles a la hora de invertir el orden de las cosas. En el escenario más popular, la cima de la pila almacena el registro de activación correspondiente al método actualmente activo. Cuando se invoca el método *G*, se inserta en la pila un registro de activación para *G*, lo que hace que *G* sea el método actualmente activo. Cuando un método vuelve, se extrae de la pila un registro y el registro de activación que pasa a estar en la parte superior de la pila contendrá los valores restaurados.

Las secuencias de llamadas a métodos y de retornos de métodos son operaciones con una pila.

Por ejemplo, la Figura 7.5 muestra una pila de registros de activación que se forma durante el proceso de evaluación de `s(4)`. En este punto, tendremos las llamadas a `main`, `s(4)` y `s(3)` pendientes y estaremos procesando activamente `s(2)`.

El gasto de espacio en el que incurrimos será la memoria utilizada para almacenar un registro de activación por cada método actualmente activo. Así, en nuestro ejemplo anterior en el que `s(8883)` fallaba, el sistema dispone de aproximadamente 8.883 registros de activación. (Observe que `main` genera por sí mismo un registro de activación.) La tarea de insertar y extraer de la pila interna un registro de activación representa también el gasto adicional correspondiente a la ejecución de una llamada a método.

La estrecha relación existente entre la recursión y las pilas nos sugiere que los programas recursivos siempre pueden implementarse de forma iterativa con una pila explícita. Presumiblemente, nuestra pila almacenará elementos más pequeños que un registro de activación, por lo que podemos confiar razonablemente en que emplearemos menos espacio. El resultado de utilizar una pila explícita es un código ligeramente más largo, pero más rápido. Los compiladores modernos con optimización de compilación han reducido los costes asociados con la recursión a un grado tal, que en lo que respecta a la velocidad, raramente merece la pena eliminar la recursión de una aplicación que la esté utilizando correctamente.

La recursión siempre puede ser eliminada utilizando una pila. Esto es necesario en ocasiones para ahorrar espacio.

7.3.4 Demasiada recursión puede ser peligrosa

En este texto vamos a proporcionar muchos ejemplos de la potencia de la recursión. Sin embargo, antes de examinar esos ejemplos, es preciso tener en cuenta que la recursión no siempre resulta adecuada. Por ejemplo, el uso de la recursión en la Figura 7.1 es desaconsejable, porque un bucle

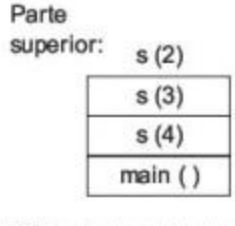

Figura 7.5 Una pila de registros de activación.

No utilice la recursión como sustituto de un bucle sencillo.

permitiría resolver el problema igualmente. Una consideración práctica a este respecto es que la gestión requerida por las llamadas recursivas consume un cierto tiempo y limita los valores de `n` para los que el programa es correcto. Una regla heurística adecuada es que jamás debe utilizarse la recursión como sustituto de un bucle sencillo.

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

Otro problema bastante más grave es el que se pone de manifiesto cuando intentamos calcular de manera recursiva los números de Fibonacci. Los *números de Fibonacci* F_0, F_1, ..., F_i se definen de la forma siguiente: $F_0 = 0$ y $F_1 = 1$; el i-ésimo número de Fibonacci es igual a la suma de los números de Fibonacci (i-ésimo $- 1$) y (i-ésimo $- 2$); por tanto $F_i = F_{i-1} + F_{i-2}$. A partir de esta definición, podemos determinar que la serie de los números de Fibonacci continua de la forma siguiente: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

No lleve a cabo trabajo redundante de manera recursiva, el programa será terriblemente ineficiente.

Los números de Fibonacci tienen una increíble serie de propiedades, que parece estar siempre incrementándose. De hecho, hay una revista, *The Fibonacci Quarterly*, creada con el único propósito de publicar teoremas relativos a los números de Fibonacci. Por ejemplo, la suma de los cuadrados de dos números de Fibonacci consecutivos es otro número de Fibonacci. La suma de los N primeros números de Fibonacci es siempre una unidad inferior a F_{N+2} (consulte el Ejercicio 7.11 para ver algunas otras igualdades interesantes).

Puesto que los números de Fibonacci se definen de manera recursiva, parece natural escribir una rutina recursiva con el fin de determinar F_N. Esta rutina recursiva, mostrada en la Figura 7.6, funciona, pero presenta un problema grave. En nuestra máquina relativamente rápida, se tarda aproximadamente un minuto en calcular F_{40}, lo cual es una cantidad absurda de tiempo, si tenemos en cuenta que el cálculo básico solo requiere 39 sumas.

El problema subyacente es que esta rutina recursiva realiza cálculos redundantes. Para calcular `fib(n)`, calculamos recursivamente `fib(n-1)`. Cuando termina la llamada recursiva, calculamos `fib(n-2)` utilizando otra llamada recursiva. Pero ya habíamos calculado `fib(n-2)` durante el proceso de cálculo de `fib(n-1)`, así que la llamada a `fib(n-2)` es un cálculo redundante, y por tanto un desperdicio de recursos. De hecho, hemos hecho dos llamadas a `fib(n-2)` en lugar de solo una.

Normalmente, hacer dos llamadas a métodos en lugar de una representaría duplicar el tiempo de ejecución del programa. Sin embargo, en este caso, las cosas son aún peores: cada llamada a

```
1    // Calcular el n-ésimo número de Fibonacci.
2    // 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 Una rutina recursiva para calcular los números de Fibonacci: no es una buena idea.

`fib(n-1)` y cada llamada a `fib(n-2)` hacen una llamada a `fib(n-3)`; por tanto, habrá en la práctica tres llamadas a `fib(n-3)`. De hecho, el tema va empeorando según avanzamos, cada llamada a `fib(n-2)` o `fib(n-3)` provoca una llamada a `fib(n-4)`, así que habrá cinco llamadas a `fib(n-4)`. Es decir, obtenemos un efecto de composición: cada llamada recursiva va haciendo cada vez más trabajo redundante.

La rutina recursiva `fib` es exponencial.

Sea $C(N)$ el número de llamadas a `fib` realizadas durante la evaluación de `fib(n)`. Claramente, $C(0) = C(1) = 1$ llamada. Para $N \geq 2$, llamamos a `fib(n)`, más todas las llamadas necesarias para evaluar `fib(n-1)` y `fib(n-2)` de forma recursiva e independiente. Por tanto, $C(N) = C(N-1) + C(N-2) + 1$. Por inducción, podemos verificar fácilmente que para $N \geq 3$ la solución a esta recurrencia es $C(N) = F_{N+2} + F_{N-1} - 1$. Por tanto, el número de llamadas recursivas es mayor que el número de Fibonacci que estamos tratando de calcular, y crece exponencialmente. Para $N = 40$, $F_{40} = 102.334.155$, y el número total de llamadas recursivas supera los 300.000.000. No resulta extraño, por tanto, que el programa se eternice. El aumento explosivo de llamadas recursivas se ilustra en la Figura 7.7.

La cuarta regla fundamental de la recursión: nunca duplique el trabajo que hay que realizar, resolviendo la misma instancia de un problema mediante llamadas recursivas separadas.

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

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

7.3.5 Previsualización de árboles

El *árbol* es una estructura fundamental en las Ciencias de la computación. Casi todos los sistemas operativos almacenan los archivos en árboles o estructuras similares al árbol. Los árboles se emplean también en el diseño de compiladores, el procesamiento de textos y los algoritmos de búsqueda. Hablaremos en detalle de los árboles en los Capítulos 18 y 19. También haremos uso de los árboles en las Secciones 11.2.4 (árboles de expresión) y 12.1 (códigos de Huffman).

Un *árbol* consta de un conjunto de nodos y de un conjunto de aristas dirigidas que los unen.

Hay una definición de árbol que es recursiva: o bien un árbol está vacío o bien consta de una raíz y de cero o más subárboles no vacíos $T_1, T_2, \ldots, T_k$, cada una de cuyas raíces están conectadas por una arista a la raíz, como se ilustra en la Figura 7.8. En algunos casos (y especialmente en los *árboles binarios* de los que hablaremos en el Capítulo 18), podemos permitir que algunos de los subárboles estén vacíos.

Utilizando una definición no recursiva, un *árbol* está compuesto por un conjunto de nodos y un conjunto de aristas dirigidas que conectan parejas de nodos. A lo largo de este texto solo vamos a tomar en consideración los árboles que tienen raíz. Un árbol que tiene raíz presenta las siguientes propiedades:

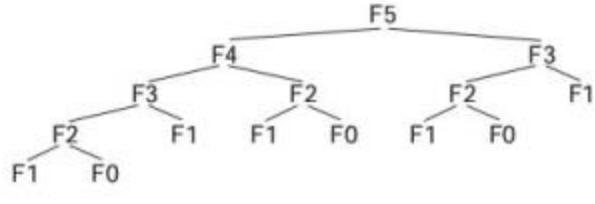

Figura 7.7 Una traza del cálculo recursivo de los números de Fibonacci.

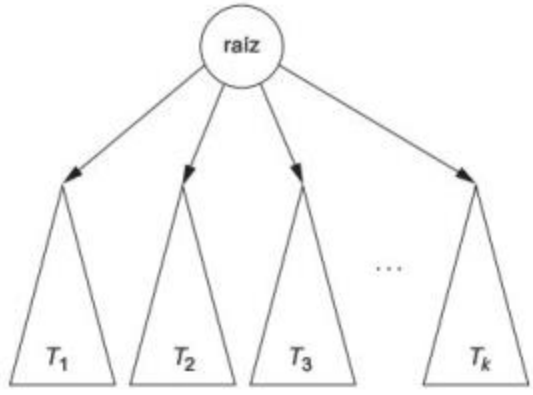

Figura 7.8 Una visión recursiva de un árbol.

- Uno de los nodos está designado como raíz.
- Todo nodo *c*, salvo la raíz, tiene una arista dirigida que entra en él y proviene de un nodo *p*. El nodo *p es el padre de c* y *c es uno de los hijos de p*.
- Existe un camino unívoco que recorre el árbol desde la raíz a cada nodo. El número de aristas que hay que recorrer es la *longitud de ese camino*.

Los padres y los hijos de definen de manera natural. Una arista dirigida conecta al *padre* con el *hijo*.

Una *hoja* no tiene ningún hijo.

Los padres y los hijos se definen de forma natural. Una arista dirigida conecta al *padre* con el *hijo*.

La Figura 7.9 muestra un árbol. El nodo raíz es *A*: los hijos de *A* son *B*, *C*, *D* y *E*. Puesto que *A* es el nodo raíz, no tiene padre; todos los restantes nodos sí tienen padre. Por ejemplo, el padre de *B* es *A*. Un nodo que no tiene ningún hijo se denomina *hoja*. Las hojas de este árbol son *C*, *F*, *G*, *H*, *I* y *K*. La longitud del camino que va desde *A* hasta *K* es 3 (aristas); la longitud del camino que va desde *A* a *A* es 0 (aristas).

7.3.6 Ejemplos adicionales

Quizá la mejor forma de comprender la recursión es analizando una serie de ejemplos. En esta sección, vamos a ver cuatro ejemplos adicionales de recursión. Los dos primeros se pueden implementar fácilmente de forma no recursiva, pero los dos últimos muestran parte de la potencia que proporciona la técnica de recursión.

Factoriales

Recuerde que *N*! es el producto de los *N* primeros enteros. Por tanto, podemos expresar *N*! como *N* veces $(N-1)!$ Combinando este hecho con el caso base, $1! = 1$, esta información nos proporciona inmediatamente todo lo que necesitamos para una implementación recursiva. Dicha implementación se muestra en la Figura 7.10.

Búsqueda binaria

En la Sección 5.6.2 hemos descrito la búsqueda binaria. Recuerde que en una búsqueda binaria, realizamos una búsqueda dentro de una matriz ordenada *A* examinando el elemento medio. Si

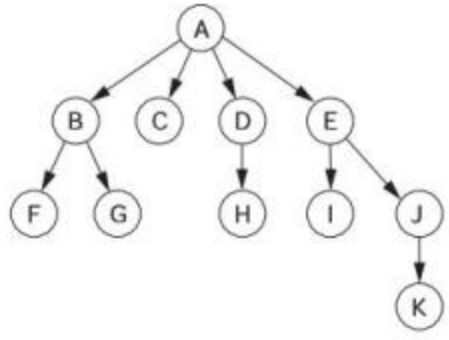

Figura 7.9 Un árbol.

encontramos una correspondencia, habremos terminado. En caso contrario, si el elemento que estamos buscando es más pequeño que el elemento medio, buscamos en la submatriz situada a la izquierda del elemento medio; en caso contrario, buscamos en la submatriz situada a la derecha del elemento medio. Este procedimiento asume que la submatriz no está vacía; si lo está, entonces el elemento no habrá podido ser encontrado.

Esta descripción se refleja directamente en el método recursivo mostrado en la Figura 7.11. El código ilustra una técnica temática en la que la rutina pública de preparación hace una llamada a una rutina recursiva y devuelve el valor de retorno de esta. Aquí, la rutina de preparación establece los puntos mínimo y máximo de la submatriz que son `0` y `a.length-1`.

En el método recursivo, el caso base de las líneas 18 y 19 se encarga de procesar la situación en la que tengamos una submatriz vacía. En caso contrario, seguimos la descripción dada anteriormente efectuando una llamada recursiva con la submatriz apropiada (líneas 24 a 26) si no se ha detectado una correspondencia. Cuando se detecta una correspondencia, se devuelve el índice respectivo en la línea 28.

Observe que el tiempo de ejecución en términos de O mayúscula, no cambia aquí con respecto a la implementación no recursiva, porque estamos realizando el mismo trabajo. En la práctica, el tiempo de ejecución sería ligeramente mayor, debido a los costes ocultos asociados con la recursión.

```
1    // Evaluar 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 7.10 Implementación recursiva del método `factorial`.

```
1   /**
2    * Realiza la búsqueda binaria estándar utilizando dos comparaciones
3    * por nivel. Esta rutina de preparación llama al método recursivo.
4    * @return el índice del elemento encontrado o NOT_FOUND si no se encuentra.
5    */
6   public static <AnyType extends Comparable<? super AnyType>>
7   int binarySearch( AnyType [ ] a, AnyType x )
8   {
9      return binarySearch( a, x, 0, a.length -1 );
10  }
11
12  /**
13   * Rutina recursiva oculta.
14   */
15  private static <AnyType extends Comparable<? super AnyType>>
16  int binarySearch( AnyType [ ] a, AnyType x, int low, int high )
17  {
18     if( low > high )
19        return NOT_FOUND;
20
21     int mid = ( low + high ) / 2;
22
23     if( a[ mid ].compareTo( x ) < 0 )
24        return binarySearch( a, x, mid + 1, high );
25     else if( a[ mid ].compareTo( x ) > 0 )
26        return binarySearch( a, x, low, mid - 1 );
27     else
28        return mid;
29  }
```

Figura 7.11 Una rutina de búsqueda binaria utilizando recursión.

Dibujo de una regla

La Figura 7.12 muestra el resultado de ejecutar un programa Java que dibuja las marcas de una regla. Aquí, consideramos el problema de marcar una pulgada. En el medio se encuentra la marca de mayor tamaño. En la Figura 7.12, a la izquierda del punto central hay una versión en miniatura de la regla y a la derecha del mismo hay una segunda versión también en miniatura de la regla. Este resultado sugiere utilizar un algoritmo recursivo que dibuje primero la línea central y luego las mitades izquierda y derecha.

No tiene que comprender los detalles de cómo se dibujan líneas y formas en Java para entender este programa. Le basta con saber que un objeto `Graphics` es algo sobre lo que se dibuja. El método `drawRuler` de la Figura 7.13 es nuestra rutina recursiva. Utiliza el método `drawLine`, que forma parte de la clase `Graphics`. El método `drawLine` dibuja una línea desde un punto de coordenadas

Figura 7.12 Una regla dibujada de forma recursiva.

(x, y) a otro punto de coordenadas (x, y), donde las coordenadas representan distancias con respecto a la esquina superior izquierda.

Nuestra rutina dibuja marcas con un número `level` de diferentes alturas; cada llamada recursiva es un nivel más profunda que la anterior (en la Figura 7.12 hay ocho niveles). Primero trata con el caso base en las líneas 4 y 5. Después se dibuja la marca central en la línea 9. Finalmente, las dos miniaturas se dibujan recursivamente en las líneas 11 y 12. En el código en línea, hemos incluido código adicional para hacer más lento el proceso de dibujo. De esta forma, se puede ver el orden en el que el algoritmo recursivo va dibujando las líneas.

Estrella fractal

En la Figura 7.14(a) se muestra un patrón aparentemente complejo denominado *estrella fractal*, el cual podemos dibujar fácilmente utilizando recursión. Todo el lienzo está inicialmente pintado de color gris (no mostrado); el patrón se forma dibujando cuadrados de color blanco sobre el fondo gris. El último cuadrado dibujado se encuentra en el centro. La Figura 7.14(b) muestra el dibujo resultante justo después de dibujar el último cuadrado. Por tanto, antes de que se dibujara el último cuadrado, se habían dibujado cuatro versiones en miniatura, una en cada uno de los cuatro cuadrantes disponibles. Este patrón nos proporciona la información necesaria para deducir el algoritmo recursivo que hay que utilizar.

Como en el ejemplo anterior, el método `drawFractal` utiliza una rutina de la librería Java. En este caso, `fillRect` dibuja un rectángulo; es preciso especificar su esquina superior izquierda y sus dimensiones. El código se muestra en la Figura 7.15. Los parámetros de `drawFractal` incluyen el centro del fractal y la dimensión global. A partir de estos datos, podemos calcular en la línea 5,

```
1    // Código Java para dibujar la Figura 7.12.
2    void drawRuler( Graphics g, int left, int right, int level )
3    {
4       if( level < 1 )
5          return;
6
7       int mid = ( left + right ) / 2;
8
9       g.drawLine( mid, 80, mid, 80 - level * 5 );
10
11      drawRuler( g, left, mid - 1, level - 1 );
12      drawRuler( g, mid + 1, right, level - 1 );
13   }
```

Figura 7.13 Un método recursivo para dibujar una regla.

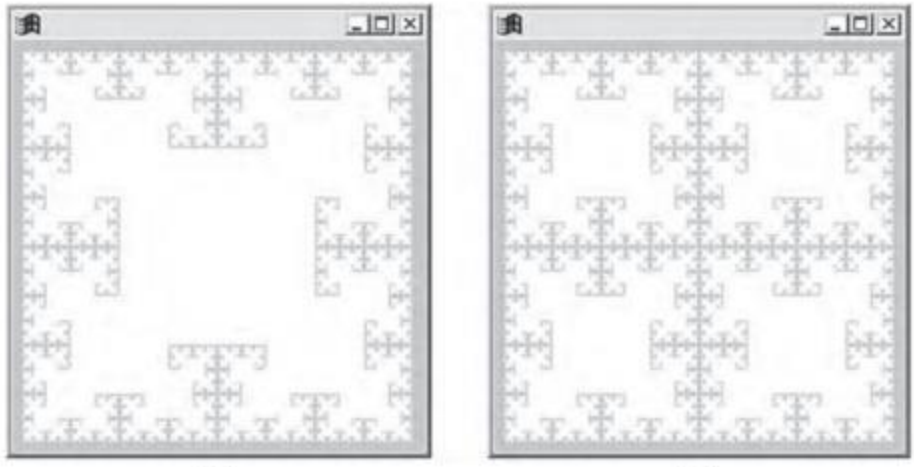

Figura 7.14 (a) Una estrella fractal dibujada mediante el código de la Figura 7.15. (b) La misma estrella inmediatamente después de añadir el último cuadrado.

```
1    // Dibujar la imagen de la Figura 7.14.
2    void drawFractal( Graphics g, int xCenter,
3                      int yCenter, int boundingDim )
4    {
5       int side = boundingDim / 2;
6
7       if( side < 1 )
8          return;
9
10         // Calcular la esquinas.
11      int left = xCenter - side / 2;
12      int top = yCenter - side / 2;
13      int right = xCenter + side / 2;
14      int bottom = yCenter + side / 2;
15
16         // Dibujar recursivamente cuatro cuadrantes.
17      drawFractal( g, left, top, boundingDim / 2 );
18      drawFractal( g, left, bottom, boundingDim / 2 );
19      drawFractal( g, right, top, boundingDim / 2 );
20      drawFractal( g, right, bottom, boundingDim / 2 );
21
22         // Dibujar el cuadrado central, donde se solapan los cuadrantes.
23      g.fillRect( left, top, right - left, bottom - top );
24   }
```

Figura 7.15 Código para dibujar la estrella fractal mostrada en la Figura 7.14.

el tamaño del gran cuadrado central. Después de gestionar el caso base en las líneas 7 y 8, calculamos las fronteras del rectángulo central. A continuación, podemos dibujar los cuatro fractales en miniatura en las líneas 17 a 20. Por último, dibujamos el cuadrado central en la línea 23. Observe que este cuadrado debe ser dibujado después de las llamadas recursivas. En caso contrario, obtendríamos una imagen distinta (en el Ejercicio 7.33, le pediremos que describa la diferencia).

7.4 Aplicaciones numéricas

En esta sección, vamos a echar vistazo a tres problemas extraídos principalmente de la teoría de números. La teoría de números se solía considerar una rama interesante, pero inútil de las matemáticas. Sin embargo, en los últimos 30 años, ha surgido una importante aplicación de la teoría de números: la seguridad de los datos. Comenzaremos nuestra exposición con unas pequeñas notas sobre fundamentos matemáticos y luego mostraremos algoritmos recursivos para resolver tres problemas. Podemos combinar estas rutinas con un cuarto algoritmo más complejo (descrito en el Capítulo 9), para implementar un algoritmo que puede emplearse para codificar y decodificar mensajes. Hasta la fecha, nadie ha sido capaz de demostrar que el esquema de cifrado aquí descrito no sea seguro.

He aquí los cuatro problemas que vamos a examinar.

1. *Exponenciación modular*: calcular $X^N \pmod{P}$.
2. *Máximo común divisor*: calcular $\gcd(A, B)$.
3. *Inversa multiplicativa*: calcular X a partir de la relación de equivalencia $AX \equiv 1 \pmod{P}$.
4. *Prueba de primalidad*: determinar si N es primo (dejaremos este problema para el Capítulo 9).

Los enteros con los que vamos a tratar son todos ellos de gran tamaño, formados por al menos 100 dígitos cada uno. Por tanto, debemos disponer de una forma de representar enteros de gran tamaño junto con un conjunto completo de algoritmos para las operaciones básicas de suma, resta, multiplicación, división, etc. Java proporciona la clase `BigInteger` con este propósito. Implementar esa clase de manera eficiente no es una tarea trivial y de hecho existe una amplia literatura técnica sobre dicha materia.

Utilizaremos números `long` para simplificar la presentación de nuestro código. Los algoritmos descritos aquí funcionan con objetos de gran tamaño, ejecutándose en una cantidad de tiempo razonable.

7.4.1 Aritmética modular

Los problemas de esta sección, así como la implementación de la estructura de datos de tabla hash (Capítulo 20), requiere el uso del operador `%` de Java. El operador `%`, al que designaremos `operator%`, calcula el resto de la división de dos tipos enteros. Por ejemplo, `13%10` da como resultado 3, igual que `3%10` y `23%10`. Cuando calculamos el resto de una división por 10, el rango de posibles resultados va de 0 a 9.[3] Este rango hace que `operator%` sea útil para generar enteros de pequeño tamaño.

[3] Si `n` es negativo, `n%10` va de 0 a −9.