6.8. Recursión

Desde una función puedes llamar a otras funciones. Ya lo hemos hecho en los ejemplos que hemos estudiado, pero ¿qué ocurriría si una función llamara a otra y ésta, a su vez,

llamara a la primera? O de modo más inmediato, ¿qué pasaría si una función se llamara a sí misma?

Una función que se llama a sí misma, directa o indirectamente, es una *función recursiva*. La recursión es un potente concepto con el que se pueden expresar ciertos procedimientos de cálculo muy elegantemente. No obstante, al principio cuesta un poco entender las funciones recursivas... y un poco más diseñar nuestras propias funciones recursivas. La recursión es un concepto difícil cuando estás aprendiendo a programar. No te asustes si este material se te resiste más que el resto.

6.8.1. Cálculo recursivo del factorial

Empezaremos por presentar y estudiar una función recursiva: el cálculo recursivo del factorial de un número natural. Partiremos de la siguiente definición matemática, válida para valores positivos de n:

$$n! = \begin{cases} 1, & \text{si } n = 0 \text{ o } n = 1; \\ n \cdot (n-1)!, & \text{si } n > 1. \end{cases}$$

Es una definición de factorial un tanto curiosa: ¡se define en términos de sí misma! El segundo de sus dos casos dice que para conocer el factorial de n hay que conocer el factorial de $n-1$ y multiplicarlo por n. Entonces, ¿cómo calculamos el factorial de $n-1$? En principio, conociendo antes el valor del factorial de $n-2$ y multiplicando ese valor por $n-1$. ¿Y el de $n-2$? Pues del mismo modo... y así hasta que acabemos por preguntarnos cuánto vale el factorial de 1. En ese momento no necesitaremos hacer más cálculos: el primer caso de la fórmula nos dice que 1! vale 1.

Vamos a plasmar esta idea en una función Python:

factorial_3.py — factorial.py

```
1 def factorial(n):
2     if n == 0 or n == 1:
3         resultado = 1
4     elif n > 1:
5         resultado = n * factorial(n-1)
6     return resultado
```

Compara la fórmula matemática y la función Python. No son tan diferentes. Python nos fuerza a decir lo mismo de otro modo, es decir, con otra *sintaxis*. Más allá de las diferencias de forma, ambas definiciones son idénticas.

Para entender la recursión, nada mejor que verla en funcionamiento. La figura 6.1 te muestra paso a paso qué ocurre si solicitamos el cálculo del factorial de 5. Estudia bien la figura. Con el anidamiento de cada uno de los pasos pretendemos ilustrar que el cálculo de cada uno de los factoriales tiene lugar mientras el anterior aún está pendiente de completarse. En el nivel más interno, *factorial*(5) está pendiente de que acabe *factorial*(4), que a su vez está pendiente de que acabe *factorial*(3), que a su vez está pendiente de que acabe *factorial*(2), que a su vez está pendiente de que acabe *factorial*(1). Cuando *factorial*(1) acaba, pasa el valor 1 a *factorial*(2), que a su vez pasa el valor 2 a *factorial*(3), que a su vez pasa el valor 6 a *factorial*(4), que a su vez pasa el valor 24 a *factorial*(5), que a su vez devuelve el valor 120.

De acuerdo, la figura 6.1 describe con mucho detalle lo que ocurre, pero es difícil de seguir y entender. Veamos si la figura 6.2 te es de más ayuda. En esa figura también se describe paso a paso lo que ocurre al calcular el factorial de 5, sólo que con la ayuda de unos muñecos.

- En el paso 1, le encargamos a Amadeo que calcule el factorial de 5. Él no sabe calcular el factorial de 5, a menos que alguien le diga lo que vale el factorial de 4.

Empezamos invocando *factorial*(5). Se ejecuta, pues, la línea 2 y como *n* no vale 0 o 1, pasamos a ejecutar la línea 4. Como *n* es mayor que 1, pasamos ahora a la línea 5. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(4). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(4).

> Invocamos ahora *factorial*(4). Se ejecuta la línea 2 y como *n*, que ahora vale 4, no vale 0 o 1, pasamos a ejecutar la línea 4. Como *n* es mayor que 1, pasamos ahora a la línea 5. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(3). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(3).
>
> > Invocamos ahora *factorial*(3). Se ejecuta la línea 2 y como *n*, que ahora vale 3, no vale 0 o 1, pasamos a ejecutar la línea 4, de la que pasamos a la línea 5 por ser *n* mayor que 1. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(2). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(2).
> >
> > > Invocamos ahora *factorial*(2). Se ejecuta la línea 2 y como *n*, que ahora vale 2, no es 0 o 1, pasamos a ejecutar la línea 4 y de ella a la 5 por satisfacerse la condición de que *n* sea mayor que 1. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(1). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(1).
> > >
> > > > Invocamos ahora *factorial*(1). Se ejecuta la línea 2 y como *n* vale 1, pasamos a la línea 3. En ella se dice que *resultado* vale 1, y en la línea 6 se devuelve ese valor como resultado de llamar a *factorial*(1).
> > >
> > > Ahora que sabemos que el valor de *factorial*(1) es 1, lo multiplicamos por 2 y almacenamos el valor resultante, 2, en *resultado*. Al ejecutar la línea 6, ése será el valor devuelto.
> >
> > Ahora que sabemos que el valor de *factorial*(2) es 2, lo multiplicamos por 3 y almacenamos el valor resultante, 6, en *resultado*. Al ejecutar la línea 6, ése será el valor devuelto.
>
> Ahora que sabemos que el valor de *factorial*(3) es 6, lo multiplicamos por 4 y almacenamos el valor resultante, 24, en *resultado*. Al ejecutar la línea 6, ése será el valor devuelto.

Ahora que sabemos que el valor de *factorial*(4) es 24, lo multiplicamos por 5 y almacenamos el valor resultante, 120, en *resultado*. Al ejecutar la línea 6, ése será el valor devuelto.

Figura 6.1: Traza del cálculo recursivo de *factorial*(5).

- En el paso 2, Amadeo llama a un hermano clónico suyo, Benito, y le pide que calcule el factorial de 4. Mientras Benito intenta resolver el problema, Amadeo se echa a dormir (paso 3).
- Benito tampoco sabe resolver directamente factoriales tan complicados, así que llama a su clon Ceferino en el paso 4 y le pide que calcule el valor del factorial de 3. Mientras, Benito se echa a dormir (paso 5).
- La cosa sigue igual un ratillo: Ceferino llama al clon David y David a Eduardo. Así llegamos al paso 9 en el que Amadeo, Benito, Ceferino y David están durmiendo y Eduardo se pregunta cuánto valdrá el factorial de 1.
- En el paso 10 vemos que Eduardo cae en la cuenta de que el factorial de 1 es muy fácil de calcular: vale 1.
- En el paso 11 Eduardo despierta a David y le comunica lo que ha averiguado: el factorial de 1! vale 1.

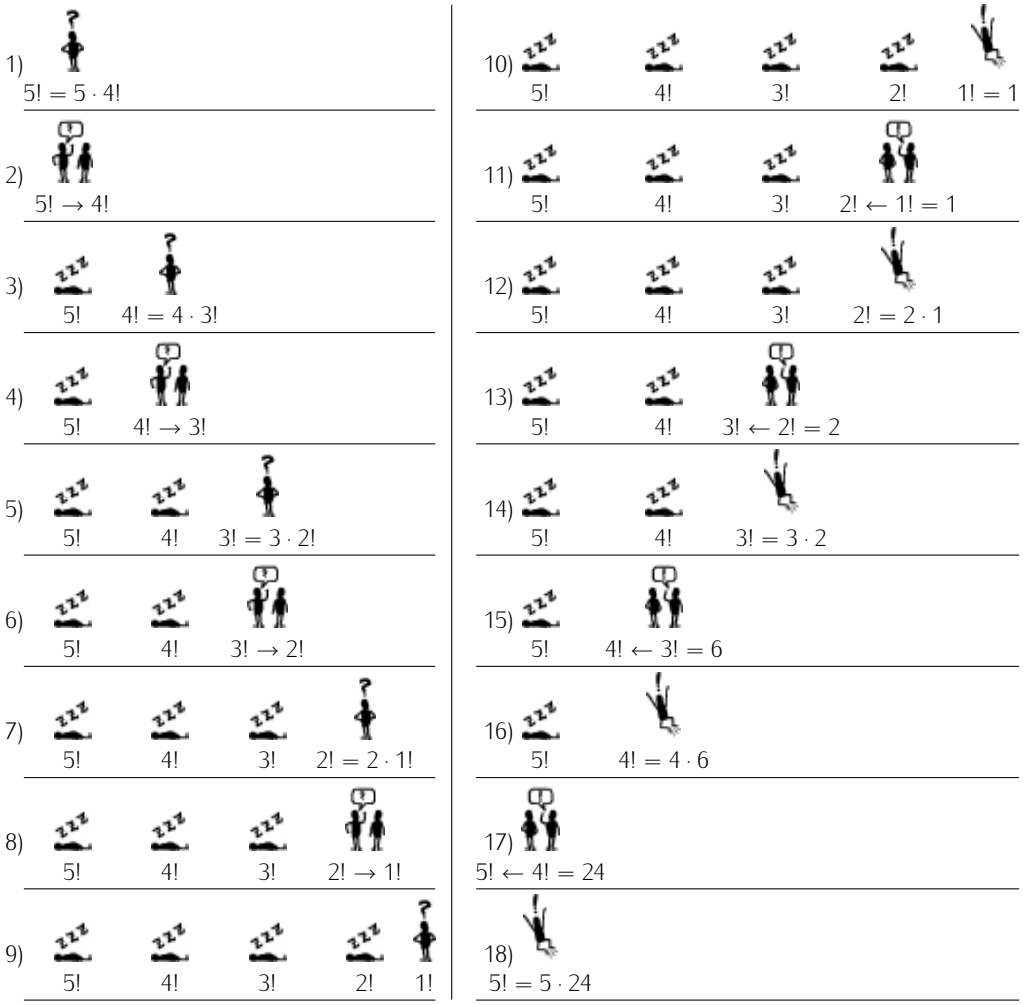


Figura 6.2: Cómic explicativo del cálculo recursivo del factorial de 5.

- En el paso 12 Eduardo nos ha abandonado: él ya cumplió con su deber. Ahora es David el que resuelve el problema que le habían encargado: 2! se puede calcular multiplicando 2 por lo que valga 1!, y Eduardo le dijo que 1! vale 1.
- En el paso 13 David despierta a Ceferino para comunicarle que 2! vale 2. En el paso 14 Ceferino averigua que 3! vale 6, pues resulta de multiplicar 3 por el valor que David le ha comunicado.
- Y así sucesivamente hasta llegar al paso 17, momento en el que Benito despierta a Amadeo y le dice que 4! vale 24.
- En el paso 18 sólo queda Amadeo y descubre que 5! vale 120, pues es el resultado de multiplicar por 5 el valor de 4!, que según Benito es 24.

Una forma compacta de representar la secuencia de llamadas es mediante el denominado *árbol de llamadas*. El árbol de llamadas para el cálculo del factorial de 5 se muestra en la figura 6.3. Los nodos del árbol de llamadas se visitan de arriba a abajo (flechas de trazo continuo) y cuando se ha alcanzado el último nodo, de abajo a arriba. Sobre las flechas de trazo discontinuo hemos representado el valor devuelto por cada llamada.

EJERCICIOS

▶ 367 Haz una traza de la pila de llamadas a función paso a paso para *factorial*(5).

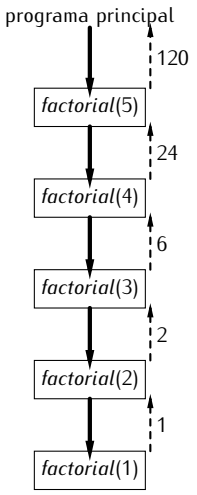


Figura 6.3: Árbol de llamadas para el cálculo de *factorial*(5).

¿Recurrir o iterar?

Hemos propuesto una solución recursiva para el cálculo del factorial, pero en anteriores apartados hemos hecho ese mismo cálculo con un método iterativo. Esta función calcula el factorial iterativamente (con un bucle **for–in**):

factorial_4.py — factorial.py

```
1 def factorial(n):
2     f = 1
3     for i in range(1,n+1):
4         f *= i
5     return f
```

Pues bien, para toda función recursiva podemos encontrar otra que haga el mismo cálculo de modo iterativo. Ocurre que no siempre es fácil hacer esa conversión o que, en ocasiones, la versión recursiva es más elegante y legible que la iterativa (o, cuando menos, se parece más a la definición matemática). Por otra parte, las versiones iterativas suelen ser más eficientes que las recursivas, pues cada llamada a una función supone pagar una pequeña penalización en tiempo de cálculo y espacio de memoria, ya que se consume memoria y algo de tiempo en gestionar la pila de llamadas a función.

▶ 368 También podemos formular recursivamente la suma de los n primeros números naturales:

$$\sum_{i=1}^{n} i = \begin{cases} 1, & \text{si } n = 1; \\ n + \sum_{i=1}^{n-1} i, & \text{si } n > 1. \end{cases}$$

Diseña una función Python recursiva que calcule el sumatorio de los n primeros números naturales.

▶ 369 Inspirándote en el ejercicio anterior, diseña una función recursiva que, dados m y n, calcule

$$\sum_{i=m}^{n} i.$$

▶ 370 La siguiente función implementa recursivamente una comparación entre dos números naturales. ¿Qué comparación?

compara.py — compara.py

```
1 def comparacion(a, b):
```

```
    if b == 0:
        return False
    elif a == 0:
        return True
    else:
        return comparacion(a-1, b-1)
```

Regresión infinita

Observa que una elección inapropiada de los casos base puede conducir a una recursión que no se detiene jamás. Es lo que se conoce por *regresión infinita* y es análoga a los bucles infinitos.

Por ejemplo, imagina que deseamos implementar el cálculo recursivo del factorial y diseñamos esta función errónea:

factorial.py

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

¿Qué ocurre si calculamos con ella el factorial de 0, que es 1? Se dispara una cadena infinita de llamadas recursivas, pues el factorial de 0 llama a factorial de −1, que a su vez llama a factorial de −2, y así sucesivamente. Jamás llegaremos al caso base.

De todos modos, el computador no se quedará colgado indefinidamente: el programa acabará por provocar una excepción. ¿Por qué? Porque la pila de llamadas irá creciendo hasta ocupar toda la memoria disponible, y entonces Python indicará que se produjo un «desbordamiento de pila» (en inglés, «stack overflow»).

6.8.2. Cálculo recursivo del número de bits necesarios para representar un número

Vamos con otro ejemplo de recursión. Vamos a hacer un programa que determine el número de bits necesarios para representar un número entero dado. Para pensar en términos recursivos hemos de actuar en dos pasos:

1. Encontrar uno o más casos sencillos, tan sencillos que sus respectivas soluciones sean obvias. A esos casos los llamaremos *casos base*.

2. Plantear el caso general en términos de un problema similar, *pero más sencillo*. Si, por ejemplo, la entrada del problema es un número, conviene que propongas una solución en términos de un problema equivalente sobre un número más pequeño.

En nuestro problema los casos base serían 0 y 1: los números 0 y 1 necesitan un solo bit para ser representados, sin que sea necesario hacer ningún cálculo para averiguarlo. El caso general, digamos n, puede plantearse del siguiente modo: el número n puede representarse con 1 bit más que el número $n/2$ (donde la división es entera). El cálculo del número de bits necesarios para representar $n/2$ parece más sencillo que el del número de bits necesarios para representar n, pues $n/2$ es más pequeño que n.

Comprobemos que nuestro razonamiento es cierto. ¿Cuántos bits hacen falta para representar el número 5? Uno más que los necesarios para representar el 2 (que es el resultado de dividir 5 entre 2 y quedarnos con la parte entera). ¿Y para representar el número 2? Uno más que los necesarios para representar el 1. ¿Y para representar el

número 1?: fácil, ese es un caso base cuya solución es 1 bit. Volviendo hacia atrás queda claro que necesitamos 2 bits para representar el número 2 y 3 bits para representar el número 5.

Ya estamos en condiciones de escribir la función recursiva:

bits_2.py — bits.py

```
def bits(n):
    if n == 0 or n == 1:
        resultado = 1
    else:
        resultado = 1 + bits(n / 2)
    return resultado
```

EJERCICIOS

▶ 371 Dibuja un árbol de llamadas que muestre paso a paso lo que ocurre cuando calculas *bits*(63).

▶ 372 Diseña una función recursiva que calcule el número de dígitos que tiene un número entero (en base 10).

6.8.3. Los números de Fibonacci

El ejemplo que vamos a estudiar ahora es el del cálculo recursivo de números de Fibonacci. Los números de Fibonacci son una secuencia de números muy particular:

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	...
1	1	2	3	5	8	13	21	34	55	89	...

Los dos primeros números de la secuencia valen 1 y cada número a partir del tercero se obtiene sumando los dos anteriores. Podemos expresar esta definición matemáticamente así:

$$F_n = \begin{cases} 1, & \text{si } n = 1 \text{ o } n = 2; \\ F_{n-1} + F_{n-2}, & \text{si } n > 2. \end{cases}$$

La transcripción de esta definición a una función Python es fácil:

fibonacci_3.py — fibonacci.py

```
def fibonacci(n):
    if n==1 or n==2:
        resultado = 1
    elif n > 2:
        resultado = fibonacci(n-1) + fibonacci(n-2)
    return resultado
```

Ahora bien, entender cómo funciona *fibonacci* en la práctica puede resultar un tanto más díficil, pues el cálculo de *un* número de Fibonacci necesita conocer el resultado de *dos* cálculos adicionales (salvo en los casos base, claro está). Veámoslo con un pequeño ejemplo: el cálculo de *fibonacci*(4).

- Llamamos a *fibonacci*(4). Como *n* no vale ni 1 ni 2, hemos de llamar a *fibonacci*(3) y a *fibonacci*(2) para, una vez devueltos sus respectivos valores, sumarlos. Pero no se ejecutan ambas llamadas simultáneamente. Primero se llama a uno (a *fibonacci*(3)) y luego al otro (a *fibonacci*(2)).
    - Llamamos primero a *fibonacci*(3). Como *n* no vale ni 1 ni 2, hemos de llamar a *fibonacci*(2) y a *fibonacci*(1) para, una vez recibidos los valores que devuelven, sumarlos. Primero se llama a *fibonacci*(2), y luego a *fibonacci*(1).

Los números de Fibonacci en el mundo real

Los números de Fibonacci son bastante curiosos, pues aparecen espontáneamente en la naturaleza. Te presentamos algunos ejemplos:

- Las abejas comunes viven en colonias. En cada colonia hay una sola reina (hembra), muchas trabajadoras (hembras estériles), y algunos zánganos (machos). Los machos nacen de huevos no fertilizados, por lo que tienen madre, pero no padre. Las hembras nacen de huevos fertilizados y, por tanto, tienen padre y madre. Estudiemos el árbol genealógico de **1** zángano: tiene **1** madre, **2** abuelos (su madre tiene padre y madre), **3** bisabuelos, **5** tatarabuelos, **8** tatara-tatarabuelos, **13** tatara-tatara-tatarabuelos... Fíjate en la secuencia: 1, 1, 2, 3, 5, 8, 13... A partir del tercero, cada número se obtiene sumando los dos anteriores. Esta secuencia es la serie de Fibonacci.
- Muchas plantas tienen un número de pétalos que coincide con esa secuencia de números: la flor del iris tiene 3 pétalos, la de la rosa silvestre, 5 pétalos, la del dephinium, 8, la de la cineraria, 13, la de la chicoria, 21... Y así sucesivamente (las hay con 34, 55 y 89 pétalos).
- El número de espirales cercanas al centro de un girasol que van hacia a la izquierda y las que van hacia la derecha son, ambos, números de la secuencia de Fibonacci.
- También el número de espirales que en ambos sentidos presenta la piel de las piñas coincide con sendos números de Fibonacci.

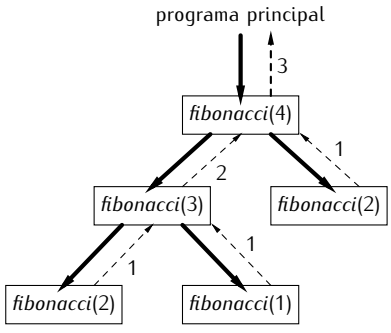
Podríamos dar aún más ejemplos. Los números de Fibonacci aparecen por doquier. Y además, son tan interesantes desde un punto de vista matemático que hay una asociación dedicada a su estudio que edita trimestralmente una revista especializada con el título *The Fibonacci Quarterly*.

  - Llamamos primero a *fibonacci*(2). Este es fácil: devuelve el valor 1.
  - Llamamos a continuación a *fibonacci*(1). Este también es fácil: devuelve el valor 1.

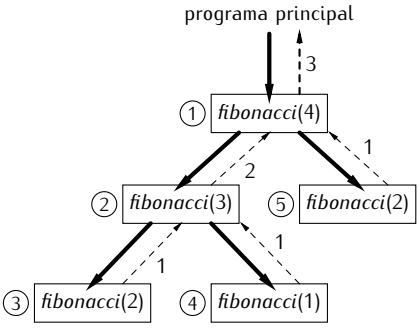
  Ahora que sabemos que *fibonacci*(2) devuelve un 1 y que *fibonacci*(1) devuelve un 1, sumamos ambos valores y devolvemos un 2. (Recuerda que estamos ejecutando una llamada a *fibonacci*(3).)
- Y ahora llamamos a *fibonacci*(2), que inmediatamente devuelve un 1.

Ahora que sabemos que *fibonacci*(3) devuelve un 2 y que *fibonacci*(2) devuelve un 1, sumamos ambos valores y devolvemos un 3. (Recuerda que estamos ejecutando una llamada a *fibonacci*(4).)

He aquí el árbol de llamadas para el cálculo de *fibonacci*(4):



¿En qué orden se visitan los nodos del árbol? El orden de visita se indica en la siguiente figura con los números rodeados por un círculo.



¿Programas eficientes o algoritmos eficientes?

Hemos presentado un programa recursivo para el cálculo de números de Fibonacci. Antes dijimos que todo programa recursivo puede reescribirse con estructuras de control iterativas. He aquí una función iterativa para calcular números de Fibonacci:

fibonacci_4.py — `fibonacci.py`

```
def fibonacci_iterativo(n):
    if n == 1 or n == 2:
        f = 1
    else:
        f1 = 1
        f2 = 1
        for i in range(3, n+1):
            f = f1 + f2
            f1 = f2
            f2 = f
    return f
```

Analízala hasta que entiendas su funcionamiento (te ayudará hacer una traza). En este caso, la función iterativa es *muchísimo* más rápida que la recursiva. La mayor rapidez no se debe a la menor penalización porque hay menos llamadas a función, sino al propio algoritmo utilizado. El algoritmo recursivo que hemos diseñado tiene un *coste exponencial*, mientras que el iterativo tiene un *coste lineal*. ¿Que qué significa eso? Pues que el número de «pasos» del algoritmo lineal es directamente proporcional al valor de n, mientras que crece brutalmente en el caso del algoritmo recursivo, pues cada llamada a función genera (hasta) dos nuevas llamadas a función que, a su vez, generarán (hasta) otras dos cada una, y así sucesivamente. El número total de llamadas recursivas crece al mismo ritmo que 2^n... una función que crece muy rápidamente con n.

¿Quiere eso decir que un algoritmo iterativo es siempre preferible a uno recursivo? No. No siempre hay una diferencia de costes tan alta.

En este caso, no obstante, podemos estar satisfechos del programa iterativo, al menos si lo comparamos con el recursivo. ¿Conviene usarlo siempre? No. El algoritmo iterativo no es el más eficiente de cuantos se conocen para el cálculo de números de Fibonacci. Hay una fórmula no recursiva de F_n que conduce a un algoritmo aún más eficiente:

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

Si defines una función que implemente ese cálculo, verás que es mucho más rápida que la función iterativa. Moraleja: la clave de un programa eficiente se encuentra (casi siempre) en diseñar (¡o encontrar en la literatura!) un algoritmo eficiente. Los libros de algorítmica son una excelente fuente de soluciones ya diseñadas por otros o, cuando menos, de buenas ideas aplicadas a otros problemas que nos ayudan a diseñar mejores soluciones para los nuestros. En un tema posterior estudiaremos la cuestión de la eficiencia de los algoritmos.

▶ 373 Calcula F_{12} con ayuda de la función que hemos definido.

▶ 374 Dibuja el árbol de llamadas para *fibonacci*(5).

▶ 375 Modifica la función para que, cada vez que se la llame, muestre por pantalla un mensaje que diga «**Empieza cálculo de Fibonacci de** *n*», donde *n* es el valor del argumento, y para que, justo antes de acabar, muestre por pantalla «**Acaba cálculo de Fibonacci de** *n* **y devuelve el valor** *m*», donde *m* es el valor a devolver. A continuación, llama a la función para calcular el cuarto número de Fibonacci y analiza el texto que aparece por pantalla. Haz lo mismo para el décimo número de Fibonacci.

▶ 376 Puedes calcular recursivamente los números combinatorios sabiendo que, para $n \geq m$,

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

y que

$$\binom{n}{n} = \binom{n}{0} = 1.$$

Diseña un programa que, a partir de un valor *n* leído de teclado, muestre $\binom{n}{m}$ para *m* entre 0 y *n*. El programa llamará a una función *combinaciones* definida recursivamente.

▶ 377 El número de formas diferentes de dividir un conjunto de *n* números en *k* subconjuntos se denota con $\left\{ {n \atop k} \right\}$ y se puede definir recursivamente así:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

El valor de $\left\{ {n \atop 1} \right\}$, al igual que el de $\left\{ {n \atop n} \right\}$, es 1. Diseña un programa que, a partir de un valor *n* leído de teclado, muestre $\left\{ {n \atop m} \right\}$ para *m* entre 0 y *n*. El programa llamará a una función *particiones* definida recursivamente.