Capítulo 3
Algoritmos recursivos

Los *algoritmos recursivos* se basan en la metodología de llamar repetidamente la propia función en que están definidos, y son de gran utilidad en multitud de campos en la informática.

Al finalizar el estudio de estas lecciones serás capaz de:

- Conocer los fundamentos y características de los algoritmos recursivos.
- Conocer el funcionamiento de las funciones recursivas.
- Conocer las características fundamentales de las estructuras de datos recursivas.
- Conocer las ventajas y desventajas de las funciones recursivas frente a las funciones iterativas.

Lección 1
Algoritmos recursivos

Introducción

Una técnica común de resolución de problemas es la división de un problema en varios subproblemas de la misma categoría, pero de más fácil resolución. Esta técnica se conoce como *divide y vencerás*.

Un ejemplo de algoritmos en los que se aplica esta técnica son los *algoritmos recursivos*, en los cuales el algoritmo se llama a sí mismo repetidamente, procurando simplificar o reducir el problema en cada llamada, hasta llegar a un caso trivial de solución directa.

DEFINICION

La *recursividad* es una técnica de programación que busca resolver un problema sustituyéndolo por otros problemas de la misma categoría, pero más simples.

La mayoría de los lenguajes soportan los algoritmos recursivos, permitiendo que una función se llame a sí misma. En un algoritmo recursivo, los bucles típicos de un algoritmo iterativo (*para..hasta, mientras…*) se sustituyen por llamadas al propio algoritmo.

DEFINICION

Se dice que un algoritmo es *recursivo* si dentro del cuerpo del algoritmo y de forma directa o indirecta se realiza una llamada a él mismo.

Al escribir un algoritmo recursivo, debe establecerse de algún modo cuando debe dejar de llamarse a sí mismo, o de otra forma se repetiría indefinidamente. Para ello, se establece una *condición de salida* llamada *caso base*.

DEFINICION

Se llama *caso base* o *condición de salida* al caso trivial de un algoritmo recursivo, del cual conocemos su solución.

El caso base contempla el caso en el cual la solución es lo suficientemente sencilla como para responder directamente sin necesidad de realizar otra llamada al algoritmo.

EJEMPLO

El ejemplo más típico de algoritmo recursivo es el de una función para calcular el factorial de un número. El factorial de un número es el resultado de multiplicar dicho número por todos los precedentes, hasta llegar a 1. Por ejemplo, factorial(3) = 3 * 2 * 1. Si observamos que el factorial de un número es equivalente al producto de dicho número por el factorial del número precedente:

*factorial(3) = 3 * factorial(2)*

podemos plantear una implementación recursiva:

```
función factorial(n)
        si n = 1
                devolver 1
        sino
                devolver n * factorial(n – 1)
        fin si
fin función
```

En este caso, la sentencia

si n = 1 devolver 1

es la *condición de salida* o *caso base* que evita que la función se llame a sí misma indefinidamente.

¡OJO!

Todo algoritmo recursivo debe incluir al menos un caso base y garantizar que se ejecuta en algún momento para evitar la recursividad infinita.

NOTA

La mayoría de los algoritmos que pueden ser descritos de forma *iterativa* (es decir, haciendo uso de bucles *while*, *for…*) pueden ser reescritos de forma *recursiva*, y viceversa.

El concepto de recursividad pone una gran potencia al alcance del programador y adquiere una dimensión fundamental en los llamados *lenguajes de programación funcionale*s, los cuales a menudo no incorporan sentencias que permitan crear bucles, debiendo recurrir a las llamadas recursivas para implementar repeticiones.

NOTA

Para obtener más información respecto a los *lenguajes de programación funcionales*, consulta la dirección Web

http://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional

Se pueden establecer diferentes categorías de recursividad en virtud de la característica del algoritmo analizada:

- Según el punto desde el cual se hace la llamada recursiva: ***recursividad directa o indirecta***.

- Según el número de llamadas recursivas efectuadas en tiempo de ejecución: ***recursividad lineal o no lineal***.

- Según el punto del algoritmo desde donde se efectúa la llamada recursiva: ***recursividad final o no final***.

NOTA

Al hablar de *funciones recursivas*, nos referimos a *algoritmos recursivos* implementados en un lenguaje particular. Aunque hemos elegido el término genérico *función*, pueden emplearse otros similares como *procedimiento* o *método*.

Recursividad directa y recursividad indirecta

▪ **Recursividad directa:** Se da cuando la función efectúa una llamada a sí misma.

```
función A
…
A(…)
…
fin función
```

▪ **Recursividad indirecta:** Se da cuando una función A llama a otra función B la cual a su vez, y de forma directa o indirecta, llama nuevamente a A.

```
función A
…
B(…)
…
fin función
```

```
función B
…
A(…)
…
fin función
```

Recursividad lineal y recursividad no lineal

▪ **Recursividad lineal o simple:** Se da cuando la recursividad es *directa* y además cada llamada a la función recursiva sólo hace una nueva llamada recursiva.

```
función A
…
A(…)
…
fin función
```

▪ **Recursividad no lineal o múltiple:** La ejecución de una llamada recursiva da lugar a más de una llamada a la función recursiva.

```
función A
…
si condición
        A(…)
fin si
…
A(…)
…
fin función
```

La recursividad será no lineal si se cumple *condición*, pues en tal caso se producen dos llamadas recursivas.

Recursividad final y recursividad no final

▪ **Recursividad final:** Se da cuando la llamada recursiva es la última operación efectuada en el cuerpo de la función. (sin tener en cuenta la sentencia *devolver*)

```
función A
        …
        devolver A(…)
fin función
```

No existe ningún procesamiento posterior a la llamada a A().

▪ **Recursividad no final:** Se da cuando la llamada recursiva no es la última operación realizada dentro de la función (sin tener en cuenta la sentencia *devolver*)

```
función A
        …
        devolver A(…) + 5
fin función
```

La llamada a la función recursiva no es la última operación efectuada (en ambos casos es una suma)

```
función A
        …
        devolver A(…) + A(…)
fin función
```

NOTA

Explicaremos más a fondo la recursividad final en la Lección 4 – Funciones recursivas finales.

EJEMPLO

La *secuencia de Fibonacci*, definida por la fórmula

fib(n) = n si n = 0 o si n = 1

fib(n) = fib(n – 2) + fib(n – 1) si n >= 2

es un ejemplo de recursividad múltiple y no final. Para el caso n>=2 se producen dos llamadas recursivas, y al regresar de la llamada recursiva el resultado no se devuelve tal cual sino que se suma con el resultado de otra llamada recursiva.

Diseño de funciones recursivas

Para escribir un algoritmo de forma recursiva es necesario intentar transformar el problema en otro similar pero más simple, así como encontrar una solución directa para los casos triviales.

Es necesario, pues:

- **Identificar y formular el *caso base*** o condición de salida del cual conocemos la solución directamente.

- **Formular el caso general** que debe poder resolverse en función del *caso base* y una transformación del caso general hacia uno más sencillo.

Lección 2
Mecanismo de recursividad

La pila de llamadas

Para comprender el funcionamiento de un algoritmo recursivo es fundamental conocer como funciona la *pila de llamadas* de un programa en ejecución.

DEFINICION

La *pila de llamadas* (*call stack* en inglés) es un segmento de memoria basado en una estructura de datos del tipo *pila* utilizada para almacenar información relacionada con las llamadas a funciones dentro de un programa.

Todo programa en ejecución tiene una pila asociada para éste propósito. En los lenguajes de alto nivel (como C o Java) la gestión de la pila de llamadas la realiza de forma automática el compilador, y el programador no necesita preocuparse de su funcionamiento.

Cuando en un punto concreto de un programa se llama a una función -sea recursiva o no- se reserva espacio en la pila para la siguiente información:

- La ***dirección de retorno de la función***:
  De modo que sea posible regresar al punto de ejecución inmediatamente posterior al de la llamada a la función.

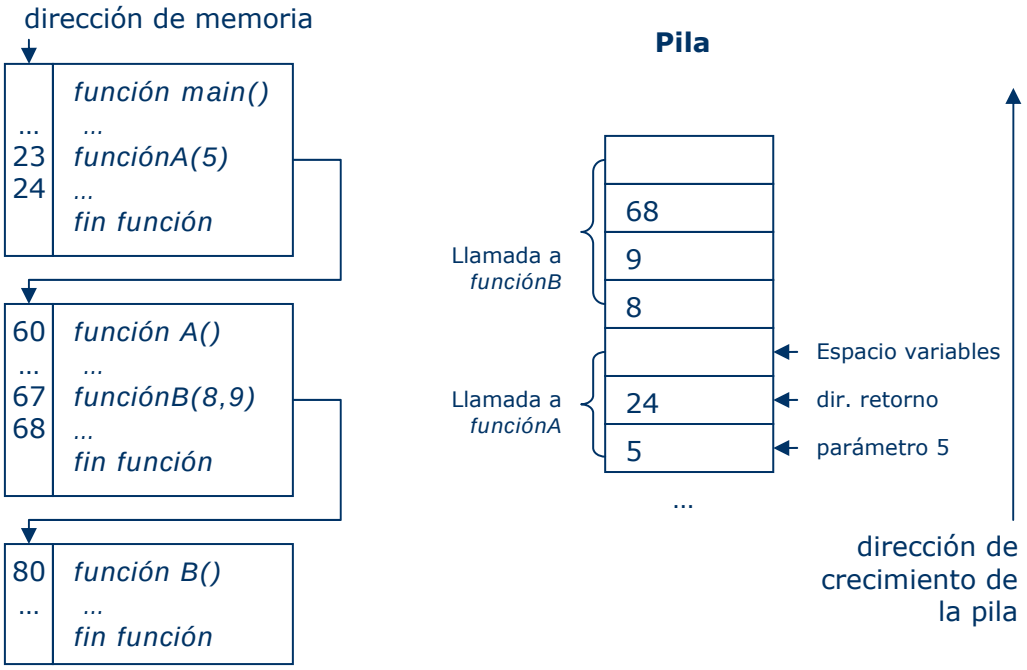
- Los ***parámetros de la llamada***:
  La función llamada obtiene los parámetros (también llamados argumentos) de la pila. Por ejemplo, en una función para sumar dos números, los argumentos serían los números a sumar.

- **Espacio para las variables locales**
  El compilador reserva espacio en la pila para almacenar las variables locales. La cantidad de espacio reservado es proporcional al número de variables locales definidas y al tamaño requerido por cada variable (carácter, entero, real…).

- El **resultado devuelto** por la función
  Opcionalmente, ya que también es posible y usual devolver el resultado de la llama a la función en uno de los *registros* de la CPU.

Dependiendo del lenguaje de programación y arquitectura de ejecución elegidos, suelen asignarse funciones extra a la pila de llamadas (por ejemplo, en los lenguajes orientados a objetos a menudo también se almacena en la pila el puntero al objeto cuyo método estamos llamando, "*this*").

En el siguiente esquema se ejemplifica la ejecución de la función *main()* de un programa. La columna *dirección de memoria* contiene la dirección de memoria (hipotética) asignada a las diferentes sentencias del programa. Conforme una función llama a otra, se almacenan los datos correspondientes en la pila:

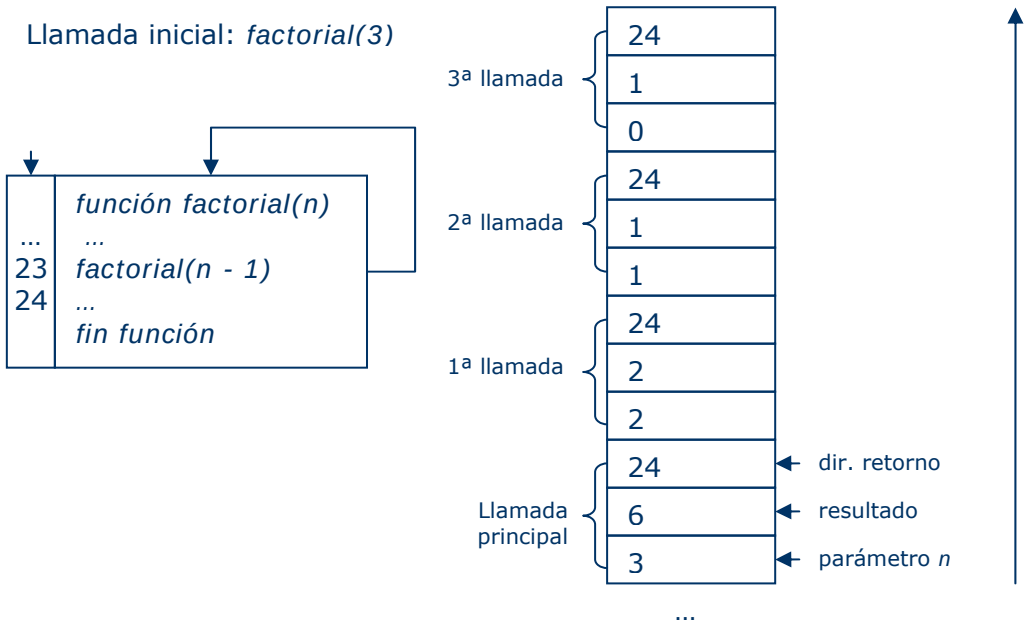


Una vez una función regresa después de su ejecución, se descarta de forma automática el espacio en la pila que se le había asignado. De esta forma, la pila va creciendo y decreciendo.

Cuanto mayor es el nivel de anidamiento de llamadas a funciones, mayor es el espacio de memoria ocupado en la pila, y existe incluso la posibilidad de ocupar todo el espacio de memoria que tiene asignado. Este hecho se conoce como *desbordamiento de pila* (*stack overflow* en inglés). Por esta razón es muy importante evitar que una función recursiva se llame indefinidamente: no sólo el programa se quedaría "colgado", sino que saturaríamos por completo la memoria destinada a la pila.

Llamadas recursivas

Una vez visto el funcionamiento básico de la pila de llamadas de un programa, podemos estudiar el uso de la pila en una función recursiva con un ejemplo. Para ello, seguiremos con el ejemplo del cálculo del factorial de un número. El siguiente esquema muestra el uso de la pila que resulta de hacer una llamada a la función *factorial*:

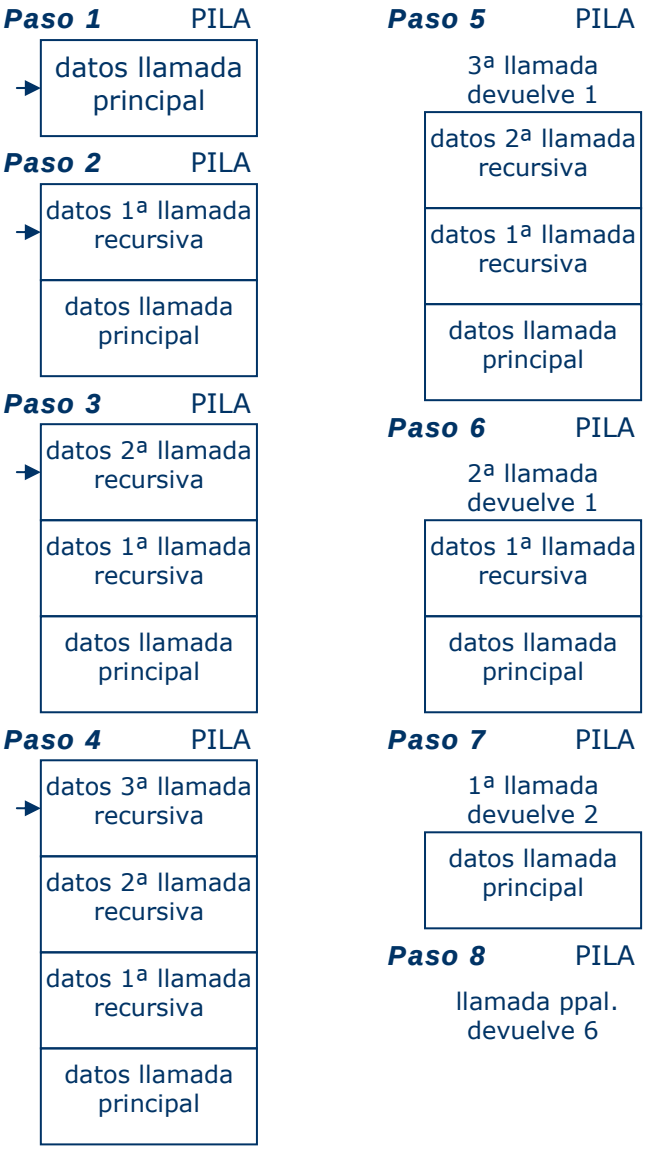


En cada llamada recursiva, se requiere espacio en la pila para almacenar *dirección de retorno*, *parámetros* y *resultado* de la llamada. La pila crece hacia arriba conforme se efectúan las llamadas: llamada principal, 1ª llamada recursiva, 2ª llamada recursiva y 3ª llamada recursiva (la final, que llega al caso base).

En el esquema anterior se han introducido en la pila los valores finales devueltos por las llamadas recursivas, pero el espacio asignado en la pila para el resultado de la llamada contiene un valor sin definir inicialmente.

La dirección de retorno es la misma en todos los casos, ya que la instrucción siguiente a cada una de las llamadas es idéntica en todos los casos.

Conforme las llamadas devuelven su resultado, se libera el espacio asignado en la pila para dicha llamada:



De este modo, tras la llamada *factorial(3)* la pila se encuentra de nuevo en el estado en el que se encontraba justo antes de la llamada a la función.

EJERCICIO

Escribe un algoritmo recursivo que calcule el elemento de cardinal más elevado de un vector de datos de tipo entero.
Por ejemplo, para la entrada {1, 3, 25, 9, 20} obtendremos como resultado 25.

EJERCICIO

El valor de x^n se puede definir recursivamente como:

$$x^0 = 1$$
$$x^n = x * x^{n-1}$$

Implementa de forma recursivo el cálculo de x^n para cualquier valor de X y n.

Lección 3
Estructuras de datos recursivas

Introducción

A menudo encontramos problemas cuya información puede presentarse de forma natural como una estructura de datos de tipo *lista*, *pila*, *cola*, *árbol* o *grafo*.

Dado que dichas estructuras de datos poseen en sí mismas carácter recursivo (pues se incluyen a si mismas en su definición), resulta natural tratar estos problemas con algoritmos recursivos.

Vamos a estudiar brevemente la aplicación del concepto de recursividad a listas enlazadas y árboles. La extensión a otros tipos de estructuras de datos es sencilla debido a su similitud (todas ellas incluyen referencias al siguiente o siguientes elementos).

Listas enlazadas

En una estructura del tipo *lista enlazada* cada *nodo* contiene una referencia al siguiente nodo (alternativamente es posible que contenga una referencia al nodo anterior, en las listas doblemente encadenadas), y un conjunto de atributos extra específicos del nodo en cuestión:

```
estructura Nodo
        dato: entero
        siguiente: Nodo
fin estructura
```

Una función que opere de forma recursiva sobre una lista enlazada, deberá procesar el nodo actual y pasar como parámetro el nodo siguiente. La condición de salida en este caso se dará cuando lleguemos a un nodo de valor *null*, que indicará el final de la lista:

```
función procesaLista(nodo)
        si nodo <> null
                imprime(nodo.dato)
                procesaLista(nodo.siguiente)
        fin si
fin función
```

Árboles

En una estructura del tipo *árbol* cada *nodo* contiene una referencia a los hijos izquierdo y derecho de la rama actual:

```
estructura Nodo
        dato : entero
        hijoIzq, hijoDch: Nodo
fin estructura
```

El algoritmo se llama pasando como parámetro inicial el nodo raíz del árbol, y las llamadas siguientes procesan los nodos hijos o subárboles del nodo principal, hasta llegar al caso base, que en un árbol corresponde a un nodo que no tiene nodos hijos, es decir, un nodo cuyos atributos *hijoIzq* e *hijoDch* son *null*. Estos nodos reciben el nombre de *hojas* del árbol. En el caso de un árbol, se procesan recursivamente tanto su lado izquierdo como su lado derecho:

```
función procesaArbol(nodo)
        si nodo <> null
                imprime(nodo.dato)
                procesaArbol(nodo.hijoIzq)
                procesaArbol(nodo.hijoIzq)
        fin si
fin función
```

Lección 4
Funciones recursivas finales

Funciones recursivas finales

En ocasiones se descarta el uso de funciones recursivas por el uso adicional de memoria que a menudo requieren respecto a la versión iterativa del mismo algoritmo. Cada llamada requiere espacio de pila para la dirección de retorno, parámetros pasados, etc.

No obstante, existe un tipo de funciones recursivas, llamadas *funciones recursivas finales,* en las cuales el tamaño de memoria de la pila utilizado no se ve afectado por el nivel de profundidad de recursión alcanzado.

DEFINICION

Se llama *función recursiva final* (*tail-call recursive function* en inglés) a toda función recursiva cuya última operación es la llamada recursiva en sí.

Para que una función recursiva sea *final* de pleno derecho, la llamada recursiva a la función debe ser la última operación realizada:

Ejemplos

De las siguientes funciones recursivas, solo la última, *recursiva3*, es recursiva final; en el resto se efectúan otras operaciones después de la llamada a la función:

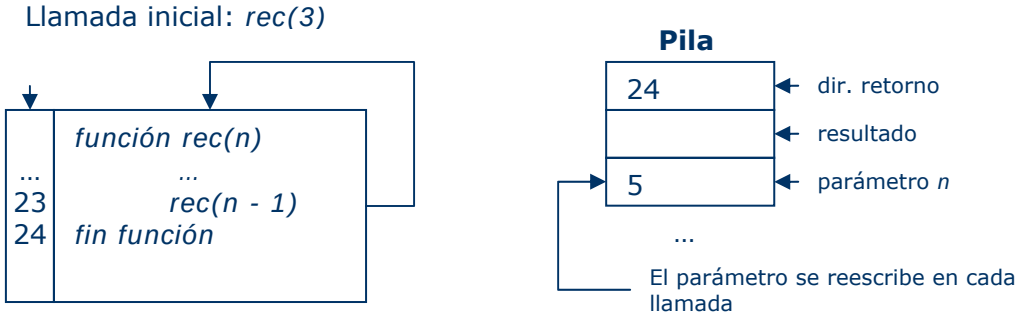
```
función recursivaA()
        num := 10
        recursivaA();
        a = a + 4;
        devolver num
fin función

función recursivaB()
        int q = 4;
        q = q + 5;
        devolver q + test1()
fin función

función recursiva3 ()
        int b = 5;
        b = b + 2;
        devolver recursiva3()
```

fin función

Tras realizar la llamada recursiva la función ya no necesita del espacio en la pila, puesto que:

- no va a hacer más uso de los parámetros pasados por la función llamante y guardados en la pila
- no va a hacer más uso de variables locales
- no va a procesar el valor de retorno

Por tanto, el compilador puede incluir código que de forma automática reutilice el espacio en pila reservado para la llamada a la función, de modo que sea utilizado por la siguiente llamada. De este modo, la penalización en tiempo que supone inicializar la pila para la siguiente llamada recursiva desaparece: Simplemente es necesario escribir los nuevos valores de los parámetros sobre los antiguos:



Con esta optimización, la eficiencia de una función recursiva se pone a la par de la versión iterativa del mismo algoritmo.

¡OJO!

No todos los compiladores soportan la optimización de funciones recursivas finales.

Lección 5
Recursividad vs. iteración

Recursividad vs Iteración

La mayoría de los algoritmos pueden expresarse tanto de forma iterativa como de forma recursiva. A la hora de implementar un algoritmo, surge la duda respecto a la conveniencia de utilizar una versión u otra.

A menudo los programadores acostumbrados al uso de *lenguajes imperativos* como C o Java descartan el uso de funciones recursivas, aludiendo a razones de eficiencia o memoria. Sin embargo, dichas deficiencias pueden minimarse a menudo.

Existen distintos factores a considerar; a continuación enumeramos los más importantes:

▪ Conveniencia

- ***por el lenguaje de programación utilizado***

  Algunos lenguajes se benefician del uso de funciones recursivas respecto a las funciones iterativas. Por ejemplo, los llamados *lenguajes funcionales*, como Lisp, Scheme o Haskell, están orientados y optimizados para el trabajo con funciones recursivas.

- ***por la definición del problema***

  Hay problemas cuya definición y/o solución son inherentemente recursivas y una solución iterativa resultaría demasiado complicada. Lo contrario también es posible.

▪ Eficiencia

En general la versión iterativa de una función siempre es más eficiente que la versión recursiva, tanto en términos de velocidad de ejecución como de memoria utilizada:

- ***Velocidad de ejecución:***

  En la versión recursiva, el hecho de que la función se llame a sí misma incurre en tiempo extra de proceso: La llamada a una función supone un coste extra en términos de tiempo de procesador en comparación con el coste que supone reiniciar una nueva iteración de un bucle: es necesario almacenar los parámetros pasados (si los hay) en la pila, así como la dirección de retorno, etc.

No obstante, si el compilador soporta las *funciones recursivas finales* y podemos escribir el algoritmo como tal, la eficiencia se pone a la par de la versión iterativa.

- ***Uso de memoria:***

En la versión recursiva, la llamada a la función requiere del uso de la *pila* para almacenar los parámetros pasados (si los hay), dirección de retorno, resultado, etc. Si el nivel de recursión puede ser muy elevado, el uso de memoria puede ser considerable o incluso prohibitivo.

No obstante, algunos algoritmos, en su versión iterativa, requieren de estructuras de datos auxiliares que son implícitas a la versión recursiva del algoritmo al pasar los datos como parámetros en la pila, de modo que consumen también mucha memoria, al nivel de la versión recursiva.

· Sencillez

En ocasiones es mucho más sencillo escribir la versión recursiva de un algoritmo que la versión iterativa, y viceversa. La elección de una u otra alternativa puede basarse en este hecho.

En general, la versión recursiva de una función es mucho más elegante y sencilla a nivel de cantidad de código requerido que la versión iterativa.

· Intuitividad

El planteamiento iterativo de una función resulta a menudo más intuitivo que el recursivo, y viceversa. Es posible favorecer la legibilidad y comprensibilidad del código respecto a la eficiencia si ésta no es un factor determinante.

NOTA

De forma general, si la eficiencia es un factor importante y/o el algoritmo se va a ejecutar de forma frecuente, conviene escribir una solución iterativa.

EJERCICIO

Reescribe el algoritmo para calcular el factorial de un número de forma iterativa.