Recursividad

Una función que se llama a sí misma se denomina **recursiva**

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

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

Utilidad

Cuando la solución de un problema se puede expresar en términos de la resolución de un problema de la misma naturaleza, aunque de menor complejidad.

Sólo tenemos que conocer la solución no recursiva para algún caso sencillo (denominado caso base) y hacer que la división de nuestro problema acabe recurriendo a los casos base que hayamos definido.

Como en las demostraciones por inducción, podemos considerar que "tenemos resuelto" el problema más simple para resolver el problema más complejo (sin tener que definir la secuencia exacta de pasos necesarios para resolver el problema).

Funcionamiento

- Se descompone el problema en problemas de menor complejidad (algunos de ellos de la misma naturaleza que el problema original).

- Se resuelve el problema para, al menos, un caso base.

- Se compone la solución final a partir de las soluciones parciales que se van obteniendo.

Diseño de algoritmos recursivos

1. Resolución de problema para los casos base:
    - Sin emplear recursividad.
    - Siempre debe existir algún caso base.

2. Solución para el caso general:
    - Expresión de forma recursiva.
    - Pueden incluirse pasos adicionales (para combinar las soluciones parciales).

Siempre se debe avanzar hacia un caso base: Las llamadas recursivas simplifican el problema y, en última instancia, los casos base nos sirven para obtener la solución.

```
int factorial (int n)
{
  int resultado;

  if (n==0)                          // Caso base
     resultado = 1;
  else                               // Caso general
     resultado = n*factorial(n-1);

  return (resultado);
}

int potencia (int base, int exp)
{
  if (exp==0)                        // Caso base
     return 1;
  else                               // Caso general
     return base * potencia(base,exp-1);
}
```

Recursividad vs. iteración

Aspectos que hay que considerar al decidir cómo implementar la solución a un problema (de forma iterativa o de forma recursiva):

- La carga computacional (tiempo de CPU y espacio en memoria) asociada a las llamadas recursivas.

- La redundancia (algunas soluciones recursivas resuelven un problema en repetidas ocasiones).

- La complejidad de la solución (en ocasiones, la solución iterativa es muy difícil de encontrar).

- La concisión, legibilidad y elegancia del código resultante de la solución recursiva del problema.

Ejemplo: Sucesión de Fibonacci

$$Fib(0) = Fib(1) = 1$$
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Solución recursiva

```
int fibonacci (int n)
{
  if ((n == 0) || (n == 1))
     return 1;
  else
     return fibonacci(n-1) + fibonacci(n-2);
}
```

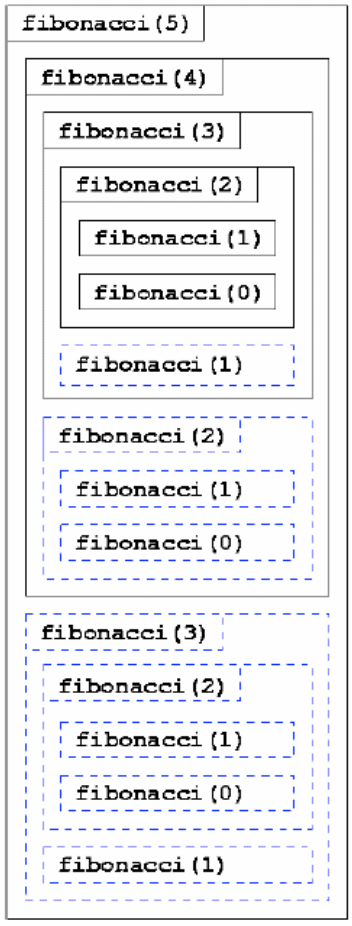
Solución iterativa

```
int fibonacci (int n)
{
  int actual, ant1, ant2;

  ant1 = ant2 = 1;

  if ((n == 0) || (n == 1)) {
     actual = 1;
  } else
     for (i=2; i<=n; i++) {
         actual = ant1 + ant2;
         ant2 = ant1;
         ant1 = actual;
     }
  }

  return (actual);
}
```



Cálculo recursivo de `fibonacci(5)`