# Recursión Versus Iteración

Maximiliano A. Eschoyez
Programación Eficiente — UBP

2010

## Caso 1: Recursión Simple

En este caso vamos a utilizar el cálculo del *factorial de un número* para comparar la eficiencia entre la versión *recursiva* y la versión *iterativa*.

La definición matemática del *factorial de un número* nos dice que:

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

La función recursiva que resuelve esto es una codificación trivial de la función matemática y podría escribirse de esta forma:

```
int Factorial (int n) {
  int f = 0;
  if ((n == 0) || (n == 1))
    f = 1;
  else if (n > 1)
    f = n * Factorial(n - 1);
  return f;
}
```

La versión iterativa podría escribirse de esta forma:

```
int Factorial (int n) {
  int f = 1;
  if (n < 0)
    f = 0;
  else if (n > 1)
    for (int i = 2; i <= n; i++)
      f *= i;
  return f;
}
```

Testee las dos versiones para diferentes valores factoriales, trace las curvas con los tiempos medidos y responda las siguientes preguntas:

1. ¿Cuál de las dos es más eficiente?
2. ¿Cuál de las dos es más programable?
3. ¿El comportamiento de ambas versiones es lineal o exponencial?

## Caso 2: Recursión Doble

En este caso vamos a utilizar el cálculo de los *Números de Fibonacci* para comparar la eficiencia entre la versión *recursiva* y la versión *iterativa*.

La definición matemática de los *Números de Fibonacci* nos dice que:

$$F_n = \begin{cases} 1 & \text{para } n = 1 \text{ y } n = 2 \\ F_{n-2} + F_{n-1} & \text{para } n > 2 \end{cases}$$

La función recursiva que resuelve esto es una codificación trivial de la función matemática y podría escribirse de esta forma:

```
int Fibonacci (int n) {
  int f = 1;
  if (n < 0)
    f = -1;
  else if (n > 2)
    f = Fibonacci(n - 2) + Fibonacci(n - 1);
  return f;
}
```

La versión iterativa podría escribirse de esta forma:

```
int Fibonacci (int n) {
  int f = 1, anterior = 1, actual = 1;
  if (n < 0)
    f = -1;
  else if (n > 2)
    for (int i = 3; i <= n; i++) {
      f = anterior + actual;
      anterior = actual;
      actual = f;
    }
  return f;
}
```

Testee las dos versiones para diferentes valores de $n$, trace las curvas con los tiempos medidos y responda las siguientes preguntas:

1. ¿Cuál de las dos es más eficiente?
2. ¿Cuál de las dos es más programable?
3. ¿El comportamiento de ambas versiones es lineal o exponencial?

### Caso 3: Combinaciones

En este caso se debe explorar la implementación del cálculo de las *Combinaciones sin repetición*, cuya definición matemática es:

$$C(n, r) = \left( \begin{array}{c} n \\ r \end{array} \right) = \frac{n!}{r!(n-r)!} \qquad \text{para } n \geq r$$

donde

$$\left( \begin{array}{c} n \\ n \end{array} \right) = \left( \begin{array}{c} n \\ 0 \end{array} \right) = 1$$

Se puede modificar la expresión matemática para que quede expuesta una versión recursiva:

$$\left( \begin{array}{c} n \\ r \end{array} \right) = \left( \begin{array}{c} n-1 \\ r \end{array} \right) + \left( \begin{array}{c} n-1 \\ r-1 \end{array} \right)$$

En este caso se están presentando la versión iterativa de la solución como así también un método matemático equivalente para solucionar el problema. Se pide analizar los siguientes casos:

1. Combinaciones con Factorial Recursivo,
2. Combinaciones con Factorial Iterativo,
3. Combinaciones versión Recursiva.

### Nota

Para la realización de este estudio se aconseja desarrollar *shell scripts* que le permitan la repetibilidad de las pruebas a realizar.

Se pide que se midan los tiempos de ejecución de cada versión para diferentes valores de entrada. También se pide que se estudie el número de llamadas a las funciones para cada versión. Estos resultados deberán verse reflejados en gráficos comparativos y árboles de llamadas.

A la hora de entregar los resultados envíe el informe en formato PDF y los código fuente utilizados tanto de los programas como de los shell scripts.