Recursión

La recursión es cuando un concepto depende de una versión anterior de sí mismo. Si aterrizamos esta idea a ciencias de computación, la recursión la podemos describir como una función que cumple con lo siguiente:

1. Tiene casos base: tiene uno o más casos base que terminan la función.
2. Tiene pasos recursivos: la función se llama a sí misma de forma directa o indirecta, de forma tal, que en cada llamada subsecuente el problema reduzca de tamaño, hasta eventualmente converger a uno de los casos bases.

Iterar con un ciclo suele ser más eficiente en memoria y tiempo, puesto que una función recursiva necesita hacer varias llamadas de memoria a lo largo de su ejecución, esto último se traduce en una constante extra por cada llamada, dicha constante puede marcar una diferencia notable al realizar un benchmark o profiling de un programa o sistema de gran escala.

Otro punto importante es que la arquitectura de computadores que todos los dispositivos modernos suelen utilizar, *arquitectura de Von Neumann*, está pensada y optimizada para trabajar con ciclos, aunque esto último da para un curso entero para entender el por qué, como así mismo explorar otras aristas como *tail optimization* que algunos lenguajes traen como feature para optimizar las llamadas recursivas. De momento con saber que la recursión tiene un costo en memoria y tiempo superior a iterar es suficiente, no hilaremos más fino que eso en el curso.

No obstante, la recursividad suele brillar y destacar en escenarios donde el problema a resolver tiene una estructura recursiva (problemas combinatorios), en estructuras de datos con definición recursiva (nodos se interconectan entre sí, mediante una estructura recursiva) o técnicas de diseño de algoritmo como divide and conquer. En dichos escenarios la recursión suele entregar soluciones simples y elegantes.

Factorial

Partamos con un ejemplo que conocen bien, el factorial de n.

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \end{cases}$$

Es una función, tiene caso base cuando n llega a cero, se llama a sí misma y en cada llamada reduce el problema en uno; multiplica el n actual por el factorial de n -1, por lo tanto para cualquier n mayor o igual que cero, podrá calcular el valor de n factorial, puesto que eventualmente llegará a 0. Es decir, es una función recursiva.

Una implementación en Java de la función factorial podría ser la siguiente:

```java
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Listing 1: Factorial en Java

Para entender la ejecución de la función factorial, hay que entender cómo se acumulan las llamadas de funciones. Los lenguajes de programación instancian las llamadas de funciones en una región de memoria llamada stack, esta región almacena cada llamada en orden LIFO hasta que son **resueltas**, la tabla de a continuación muestra la secuencia de llamadas al ejecutar factorial de 3.

Llamada	**n**	**Operación**	**Stack**
factorial(3)	3	3× factorial(2)	[3]
factorial(2)	2	2× factorial(1)	[3, 2]
factorial(1)	1	1× factorial(0)	[3, 2, 1]
factorial(0)	0	return 1	[3, 2, 1, 0]
Resolución de las llamadas:			
factorial(0)	0	1	[3, 2, 1]
factorial(1)	1	$1 \times 1 = 1$	[3, 2]
factorial(2)	2	$2 \times 1 = 2$	[3]
factorial(3)	3	$3 \times 2 = 6$	[]

Table 1: Llamadas recursivas para factorial de 3

Para analizar el tiempo de ejecución de la función factorial podemos hacer lo siguiente:

$$\begin{aligned} T(0) &= 1 \\ T(N) &= T(N-1) + c \end{aligned}$$

Donde c, simboliza el costo constante de realizar operaciones aritméticas, además del costo de cada llamada del método factorial. Por otro lado la expresión $T(N-1)$, indica que en cada llamada recursiva, el problema decrece en 1.

La pregunta del millón es cómo resolvemos la expresión $T(N)$ propuesta anteriormente. La respuesta es que hay muchas formas, pero este caso se puede resolver de manera directa con el método de expansión:

1. Primero expandimos la expresión hasta llegar al caso base:

$$\begin{aligned} T(N) &= T(N-1) + c \\ T(N-1) &= T(N-2) + c \\ T(N-2) &= T(N-3) + c \\ &\vdots \\ T(0) &= 1 \end{aligned}$$

2. Si observamos con detenimiento, la constante c se repite en cada llamada, por lo que podríamos afirmar la siguiente expresión:

$$\begin{aligned} T(N) &= T(N-1) + c \\ T(N) &= T(N-2) + 2c \\ T(N) &= T(N-3) + 3c \\ &\vdots \\ T(N) &= T(0) + Nc \end{aligned}$$

3. Por lo tanto:

$$\begin{aligned} T(N) &= T(N-k) + kc \\ T(N) &= T(0) + Nc \\ T(N) &= 1 + Nc \\ T(N) &= \mathcal{O}(N) \end{aligned}$$

Fibonacci

Ahora un segundo ejemplo, bastante conocido igualmente, Fibonacci:

$$\begin{cases} f(0) = 0 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2) \end{cases}$$

```
1  public static int fibonacci(int n) {
2      if (n == 0) {
3          return 0;
4      }
5      if (n == 1) {
6          return 1;
7      }
8      return fibonacci(n - 1) + fibonacci(n - 2);
9  }
```

Listing 2: Factorial en Java

Si ahora analizamos este algoritmo, llegaremos a lo siguiente:

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ T(N) &= T(N-1) + T(N-2) + c \end{aligned}$$

Dicha expresión necesita un poco más de análisis antes de resolverla, en casos así siempre es bueno partir con el árbol de recursión, como el siguiente:

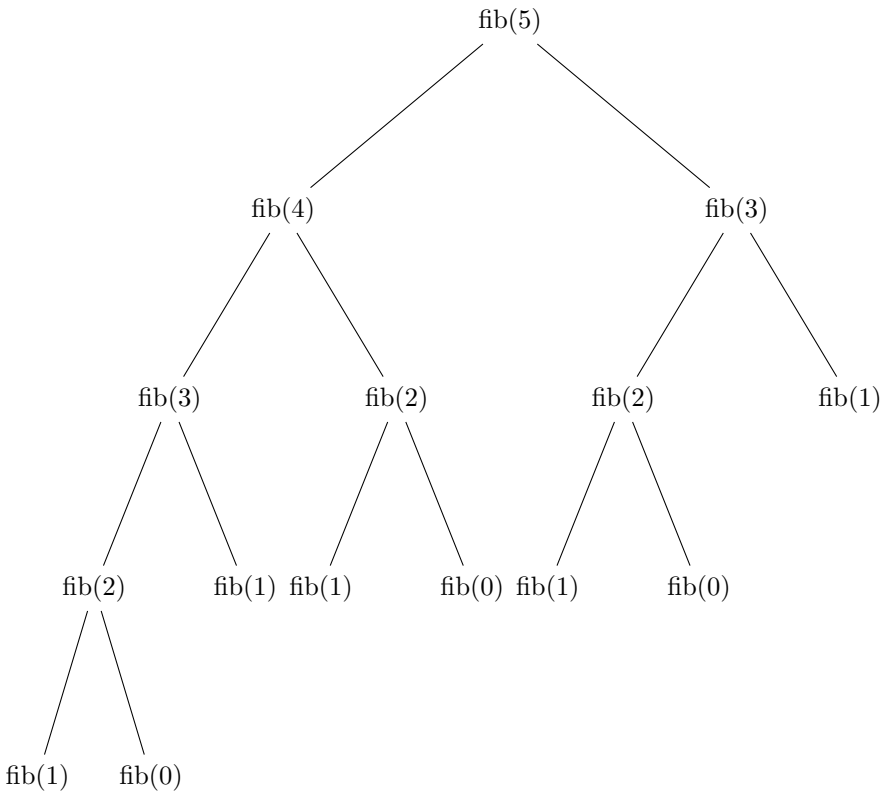

Figure 1: Árbol de recursión para Fibonacci de 5

En el árbol podemos ver que la profundidad máxima es N y que en cada nivel se van duplicando las llamadas. Con esa información podemos afirmar que Fibonacci tiene crecimiento exponencial, es decir: $T(N) \leq c * 2^N$ en el peor de los casos. Por lo que podemos asumir que $T(N) = \mathcal{O}(2^N)$, mediante el análisis del árbol de recursión. En la práctica, existe una cota aún más ajustada al crecimiento de Fibonacci $\mathcal{O}(\phi^N)$, donde ϕ es el número áureo (1.6180339...), pero un análisis de esa índole es materia de un curso más avanzado, de momento para casos así haremos uso del árbol de recursión como apoyo.