## TEMA 2: COMPLEJIDAD DE ALGORITMOS

**¿Qué entendemos por complejidad de un programa?**

Es una medida **ABSTRACTA** del <u>tiempo de ejecución</u> (conocida como **complejidad temporal**) o <u>uso de memoria</u> (conocida como **complejidad espacial**). Por norma general estas dos complejidades son inversas: (- *tiempo implica* + *espacio, y viceversa*).

La complejidad se mide en función de los datos de entrada (tamaño de la entrada o **input size**). Al crecer el tamaño de la entrada, la complejidad del programa puede crecer o permanecer constante.

Un ejemplo posible de la vida real podría ser organizar una fiesta en mi casa, teniendo en cuenta el número de invitados.

(1) ¿El tiempo que necesito para limpiar la casa antes de la fiesta depende del número de invitados?
**NO** → el trabajo es *constante* → `f(n) = c`

(2) ¿El tiempo que necesito para limpiar la casa después de la fiesta depende del número de invitados?
Probablemente **SÍ**…

(3) Quiero mandar una invitación personal a cada invitado, ¿depende del número de invitados?
**SÍ** → el trabajo es *lineal* con respecto al número de invitados → `f(n) = i*n + c`, donde `i` es el tiempo que tardo en escribir la invitación, `n` el número de invitaciones que debo escribir y `c` el tiempo que tardo en ir a correos a mandarlas

(4) Una vez en la fiesta, debemos saludarnos todos entre sí, ¿depende del número de invitados?
**SÍ** → el trabajo es *cuadrático* con respecto al número de invitados → $f(n) = s*(n^2+n)/2$, donde n es el número de invitados y s el tiempo que tarda en realizarse un saludo

**Análisis experimental:** se hace un estudio estadístico de un programa concreto, en un entorno de ejecución concreto, con un compilador concreto, en un sistema operativo concreto…

**Análisis teórico:** se estudia los programas independientemente del entorno de ejecución, analizando algoritmos en pseudocódigo o programas concretos con el objetivo de obtener los órdenes de magnitud de la complejidad. **<u>NOTA</u>:** Las constantes no son relevantes a la hora de obtener la complejidad asintótica. Además, pueden ser útiles a la hora de extrapolar los resultados teóricos a entornos concretos.

Teniendo en cuenta que cada operación (+, *, -, =, if, method call, etc) consume una unidad de tiempo (t) y que cada acceso a la memoria consume una unidad de tiempo, vamos a ver cuál es la complejidad a través de algunos ejemplos:

(1) ¿Cuánto cuesta en unidades de tiempo una llamada al método `max(1,3)`?

```
int max(int i, int j) {
  if (i > j)
      return i;
  else
      return j;
}
```

(2) ¿Cuánto cuesta en unidades de tiempo una llamada al método `member(5,[1,2,3])`?

```
static <E> boolean member(E e, E[] arr) {
  boolean found = false;
  for (int i=0; i<arr.length && !found ; i++)
    found = e.equals(arr[i]);
  return found;
}
```

**Estudio de casos:** no solo el tamaño de los datos es relevante, también la distribución de los datos puede ayudar (o perjudicar). Podemos encontrarnos diferentes escenarios para `member(E e, E[] arr)`.
El elemento a buscar…
(1) está siempre el primero;
(2) puede estar en cualquier sitio (y todos los casos son equiprobables);
(3) está siempre el último (o no está);
(4) está más veces el primero que en otra posición…

Por ello, la complejidad puede ser estudiada de muchas formas:
(1) En el caso mejor (**lower-bound**)
(2) En el caso (pro-)medio
(3) En el caso peor (**upper-bound**) → es el que nos aporta resultados más precisos y fiables, ya que nos permite movernos en un escenario "seguro".
(4) En el caso amortizado

<u>**NOTA**</u>: La complejidad asintótica implica calcular la complejidad cuando el tamaño de los datos de entrada tiende a infinito

**Funciones de complejidad:** Los interfaces no implican ninguna medida de complejidad. La complejidad va asociada a la implementación de los métodos de la interfaz en una clase. <u>**OJO**</u>: En la especificación del interfaz se puede exigir que ciertos métodos se implementen con una complejidad determinada.

*Función constante* → f (n) = c

*Función polinomial* → f (n) = $c_1 * n^{e_1} + ... + c_m * n^{e_m}$, donde el grado del polinomio lo marca el exponente de mayor valor. Entre las funciones polinomiales encontramos la lineal (grado 1), cuadrática (grado 2) y cúbica (grado 3)
*Función logarítmica* → f (n) = $log_2$ (n)

*Función exponencial* → f (n) = $c^n$

**Notación O():** el objetivo intuitivo es establecer la complejidad de una función en términos de n con una función proporcional que la acota asintóticamente ignorando los factores constantes y de orden menor. Decimos que un método tiene un orden de complejidad O(n) cuando O(n) acota asintóticamente la función de complejidad del método

Escala de complejidad de menor a mayor:

| | |
|---|---|
| Constante | O(1) |
| Logarítmica | O(log (n)) |
| Lineal | O(n) |
| N-Log-N | O(n * log (n)) |
| Cuadrática | $O(n^2)$ |
| C ubica | $O(n^3)$ |
| Polinomial de orden k | $O(n^k)$ |
| Exponencial | $O(2^n) \ldots O(m^n)$ |

**Ejemplos:**

| | |
|---|---|
| 5n + 12 | O() |
| 109 | O() |
| $n^2 + 3n + 112$ | O() |
| $n^3 + 1999n + 1337$ | O() |
| $n + \sqrt{n}$ | O() |
| $2^n + n^2$ | O() |
| n log n + 1000n + 3 | O() |

**Ejercicios:** indicar la complejidad de los siguientes métodos:

```
<E> boolean m1(IndexedList<E> l, E e) {
  boolean res = false;
  if (!l.isEmpty())
    res = l.get(0).equals(e);
  return res;
}
```

**Complejidad →**

```
void m2(IndexedList<E> l) {
  int i = 0;
  while (i < l.size()) {
    int j = 0;
    while (j < l.size())
      ++j;
    ++i;
  }
}
```

**Complejidad →**

```
<E> int method(IndexedList<E> l) {
  int i = l.size();
  int counter = 0;
  while (i > 0) {
    counter++;
    i /= 2;
  }
  return counter;
}
```

**Complejidad →**

```
<E> int method(IndexedList<E> l) {
  int counter = 0;
  for (int i=0; i < l.size(); i++) {
    int j = l.size();
    while (j > 0) {
      counter++;
      j /= 2;
    }
  }
  return counter;
}
```

**Complejidad →**