# Notación O – Grande

Cuando implementamos una nueva clase, discutimos la eficiencia de sus métodos y los comparamos con métodos similares de otras clases.

Cuando ejecutamos un programa (botón Run) la operativa es la siguiente:

- El sistema carga la JVM.
- La JVM carga el archivo .class, luego carga otros archivos .class a los que hace referencia el programa y finalmente ejecuta.

Si los archivos .class aún no se han creado, el IDE, compilará el archivo fuente antes de ejecutar. En estos dos pasos se invierte la mayor parte del tiempo de ejecución. Si se vuelve a ejecutar, inmediatamente después, puede tardar menos debido a que los archivos pueden estar aún en la memoria caché. Sin embargo, si tiene un tamaño suficientemente grande o se trata de un problema complejo, el tiempo real de ejecución del programa dominará sobre los otros tiempos comentados.

# Notación O - Grande

Por todos estos motivos es bastante difícil obtener una medida precisa del rendimiento de un algoritmo o programa.
Normalmente trataremos de aproximar el efecto de un cambio en el nº de datos que un algoritmo procesa.
De esta manera, podemos ver cómo aumenta el tiempo de ejecución de un algoritmo con respecto a n.
Podremos comparar dos algoritmos examinando sus tasas de crecimiento.
Para muchos problemas hay soluciones algorítmicas obvias pero ineficientes. En otras circunstancias son soluciones muy eficientes pero ininteligibles.
Según el caso se tratará de llegar a una solución de compromiso (legible y lo más eficiente)
Cada día las computadoras son más rápidas y con memorias más grandes. Aún así hay algoritmos cuya tasa de crecimiento es tan grande que ninguna computadora no importa cuán rápida o cuánta memoria tenga, puede resolver el problema por encima de cierto tamaño.
Por ello, es importante tener una idea de la eficiencia de los diferentes algoritmos.
Será de lo que nos ocupemos a continuación

Ejemplo 1. Búsqueda secuencial en un array de enteros

Si la clave/objetivo no está en el array (búsqueda sin éxito) el cuerpo del bucle se ejecutará (x.length) veces.

Si la clave/objetivo está podría estar en cualquier lugar, considerando el promedio de todos los casos el bucle se ejecutará (x.length/2) veces. Por lo tanto, el tiempo real de ejecución es directamente proporcional al nº de componentes, si duplicásemos el tamaño esperaríamos que se duplicase el tiempo.

Ejemplo 2 Queremos averiguar si dos arrays tienen elementos comunes. Podemos usar nuestro método de búsqueda para buscar en un array los elementos que están en el otro.

En este caso, el cuerpo del bucle se ejecutará como máximo x veces. En cada iteración ~~ho~~ invocará al método buscar anterior para buscar el elemento x[i] en el array y. El cuerpo del bucle en la búsqueda se ejecutará a lo sumo la longitud de y por lo tanto el tiempo de ejecución será x.length ~~*~~ y y.length

**Ejemplo 3** Consideremos el método que determine si cada elemento del array es único. No hay duplicados.

Si todos los valores son únicos, el bucle externo se ejecutará x.length-veces. Para cada iteración del bucle externo, el bucle interno se ejecutará también x.length-veces por tanto, el nº total de veces que se ejecutará será ~~x.len~~ $(x.\text{length})^2$

**Ejemplo 4** El método del ejemplo anterior es ineficiente porque lo hacemos tantas veces como sea necesario. Podemos reescribirlo.

Podemos inicializar el bucle interno en i+1 porque ya hemos determinado que los elementos anteriores son únicos. La 1º vez el bucle interno se ejecutará x.length-1 veces, la 2º x.length-2 veces y así sucesivamente, la última vez se ejecutará solo 1 vez. El nº total de veces que se ejecutará es:

$$x.\text{length}-1 + x.\text{length}-2 + \cdots + 2 + 1$$

La serie $1+2+3+ \cdots + (n-1)$ es la serie que tiene el valor

$$(n \times (n-1))/2 = (n^2 - n)/2$$

$$((1^{\underline{o}}\ \text{término} + \text{último}) * n^{\underline{o}}\ \text{términos})/2$$

Por tanto

$$((x.\text{length})^2 - x.\text{length})/2$$

# NOTACIÓN O-GRANDE

El tipo de análisis que acabamos de hacer es más importante para el desarrollo de software que medir los milisegundos en que se ejecuta un programa en una computadora particular. Comprender cómo crece el tiempo de ejecución (y los requisitos de memoria) de un algoritmo a medida que aumenta el tamaño de entrada (obtener la función de tamaño frente a tiempo) proporciona a los programadores una herramienta para comparar varios algoritmos.

Los informáticos han desarrollado una terminología útil y una notación para investigar y describir la relación entre el tamaño de entrada y el tiempo de ejecución.

Por ejemplo, si el tiempo se duplica cuando la entrada $n$ se duplica, decimos que tiene una tasa de crecimiento de orden $n$. Sin embargo si el tiempo se cuadruplica cuando se duplica $n$ decimos que la tasa de crecimiento ~~es de~~ tiene un orden $n^2$

~~En términos de T(n), formalmente, la notación O-Grande~~

$$\cancel{T(n) \leq O(f(n))}$$

En los ejemplos anteriores hemos analizado 4 métodos

1º _ Tiempo de ejecución relacionado con x.length
2º _ " " " " x.length * y.length
3º _ " " " " $(x.length)^2$
4º _ " " " " $(x.length)^2$ y x.length

Los informáticos utilizan la notación

1º_ $O(n)$
2º_ $O(n \times m)$
3º_ $O(n^2)$
4º_ $O(n^2)$

donde n = tamaño del problema en este caso x.length

m = y.length

El símbolo O que lo veréis en distintos tipos y estilos de letra en la literatura de Ciencias de la Computación puede considerarse la abreviatura de "Orden de Magnitud". Esta notación se llama notación O-Grande

Una manera sencilla de determinar el orden de magnitud de un algoritmo es ver si los bucles están anidados, suponiendo que el cuerpo del bucle consta de declaraciones simples:

1 solo bucle ⇒ $O(n)$
2 bucles ⇒ $O(n^2)$
3 " ⇒ $O(n^3)$ ....

Sin embargo se debe de examinar también el nº de veces que se ejecuta el bucle.

Consideremos las siguientes líneas de código:

```
for( int i = 1; i < x.length; i *= 2) {
    // Algo sobre x[i]
```

El cuerpo del bucle se ejecutará $K-1$ veces teniendo i los valores $1, 2, 4, 8, 16, \ldots\ldots, 2^{K}$ hasta $2^{K}$.

Como $2^{K-1} = x.\text{length} < 2^{K}$ sacando logaritmos

$$\log_2 2^{K} = K$$

Como $K-1 = \log_2 (x.\text{length}) < K$

entonces el orden del bucle es $O(\log n)$

La función crece lentamente. El logaritmo en base 2 de 1000.000 es aproximadamente 20.

Para analizar el tiempo de ejecución de los algoritmos utilizamos logaritmos en base 2.

## Definición formal de O-Grande

Consideremos las siguientes líneas de código.

```
for( int i=0; i<n; i++){
        for( int j=0; j<n; j++){
                    Sentencia simple
        }
}
```

for( int K=0; K<n ; K++) {

Sentencia simple 1

⋮

Sentencia simple 5

}

Sentencia simple 6

⋮

Sentencia simple 30

Supongamos que cada sentencia simple emplea una unidad de tiempo. La cabecera del for es también una unidad de tiempo. El bucle anidado ejecuta $n^2$ veces las sentencias simples. Después 5 sentencias simples se ejecutan dentro de un for n veces con la variable de control K y finalmente 25 sentencias simples.

Podríamos obtener la expresión:

$$T(n) = n^2 + 5n + 25$$

Que representaría la relación entre el tamaño o nº de elementos procesados y el tiempo de procesamiento. T(n) es el tiempo de procesamiento en función de n

Algo IMPORTANTE!! el término $n^2$ domina a medida que n se hace grande

En términos de $T(n)$, formalmente, la notación O-Grande

DEFINICIÓN

$$T(n) = O(f(n))$$

significa que existen 2 constantes, no y c mayores que cero y una función, $f(n)$, tal que para todo $n > n_0$, $c f(n) \geq T(n)$. En otras palabras, a medida que n se hace lo suficientemente grande (mayor que $n_0$) hay una cte c para la cual el tiempo de procesamiento siempre será menor o igual a $c.f(n)$, entonces $c f(n)$ es un límite superior en el rendimiento. El rendimiento nunca será peor que $c f(n)$ y puede ser mejor.

Si podemos determinar como el valor de $f(n)$ aumenta con n, sabemos como aumenta el tiempo de procesamiento con n.

La tasa de crecimiento de $f(n)$ estará determinada por la tasa de crecimiento del término más rápido de crecimiento (el que tiene mayor exponente) en este caso el término $n^2$.

Esto significa que en el ejemplo anterior el algoritmo es de orden $O(n^2)$ y podemos ignorar los términos en n inferiores y las cts dado que el análisis se hace para valores de $n \gg$

**Ejemplo 5** Dado $T(n) = n^2 + 5n + 25$ queremos mostrar que esto es $O(n^2)$ por tanto queremos mostrar que hay ctes $n_0$ y $c$ tales que para todos $n > n_0$,
$cn^2 > n^2 + 5n + 25$

Una forma de hacer esto es encontrar un punto donde

$$cn^2 = n^2 + 5n + 25$$

Si dejamos $n$ ser $n_0$ y resolvemos para $c$ obtenemos

$$c = 1 + \frac{5}{n_0} + \frac{25}{n_0^2}$$

Podemos evaluar la expresión de la derecha fácilmente cuando $n_0 = 5$

$$\Downarrow$$

$$1 + 5/5 + 25/25$$

lo cual da $\boxed{c = 3}$

Entonces $3n^2 > n^2 + 5n + 25$ para todo $n > 5$ tal y como muestra la figura 2.1.

Ejemplo 6 Consideramos el siguiente trozo o líneas de código:

```
for( int i = 0; i < n-1 ; i++ ) {
  for( int j = i+1 ; j < n ; j++ ) {
        3 sentencias simples
  }
}
```

La primera ejecución del bucle externo, el bucle interno se ejecuta n-1 veces, la siguiente n-2 ..... y la última 1.

Obtenemos entonces la siguiente expresión:

$$3(n-1) + 3(n-2) + \dots + 3(2) + 3(1)$$

Sacamos factor común de 3

$$3((n-1) + (n-2) + \dots + 2 + 1)$$

La suma del paréntesis es:

$$(n^2 - n)/2$$

Entonces $T(n) = 1.5n^2 - 1.5n$

~~Este polinomio vale cero cuando $n = 1$~~

~~Para valores mayores que 1~~

Para $n = 1$ $T(n) = 0$

Para $n > 1$ $1.5n^2$ siempre $> 1.5n^2 - 1.5n$

Por lo tanto podemos usar 1 para $n_0$ y 1.5 para $c \Rightarrow T(n)$ es $O(n^2)$ ver figura 2.2

Si $T(n)$ es la forma de un polinomio de grado **d** (el máximo exponente), entonces es $O(n^d)$

La demostración rigurosa está fuera del objetivo de este tema.

Intuitivamente puede verse con los ejemplos anteriores.

Usaremos la expresión $O(1)$ para representar una tasa de crecimiento constante

## SUMARIO DE NOTACIÓN

| Símbolo | Significado |
| --- | --- |
| $T(n)$ | Tiempo que tarda un método/programa como una función de la entrada $n$. Es posible no determinarlo exactamente. |
| $f(n)$ | Cualquier función de $n$, en general $f(n)$ representará ~~una~~ la función más simple ~~que~~ de $T(n)$ por ejemplo $n^2$ en lugar de $1.5n^2 - 1.5n$ |
| $O(f(n))$ | Orden de magnitud. $O(f(n))$ es el cj de funcions que crecen no más rápido que $f(n)$. Decimos que $T(n) = O(f(n))$ para indicar que el crecimiento de $T(n)$ es delimitado por el crecimiento de $f(n)$ |

## COMPARACIÓN DE RENDIMIENTO

Resumen tabla 2.2

La figura 2.3 muestra la tasa de crecimiento logarítmica, lineal, logarítmo lineal (n log n), cuadrática ........

Tengase en cuenta que para valores pequeños de n la función exponencial es la menor de todas no es hasta n = 20 que la función lineal es más pequeño que la cuadrática.

Esto significa que para valores pequeños de n el algoritmo menos eficiente puede ser el más eficiente (Esperemos el más claro) Si vas a procesar pocos datos el algoritmo $O(n^2)$ puede ser más apropiado que el " $O(n \log n)$ que tiene un valor cte grande (LA TIRANIA DE LA CTE)

Los algoritmos con órdenes exponenciales pueden comenzar tomando valores pequeños pero muy rapidamente se disparan.

Comentar tabla 2.3

## ALGORITMOS CON TASAS DE CRECIMIENTO EXPONENCIAL Y FACTORIAL

Si tenemos un algoritmo $O(2^n)$ que tarda 1 hora para una entrada $n = 100$, añadir una entrada $n = 101$ tardará 1 hora más, añadir 5 $n = 105$ tardará 32 horas más y añadir 14 $\Rightarrow n = 114$ tardará 16384 horas más lo cual son casi **2 AÑOS**

Esta relación es la base de los algoritmos criptográficos. Algoritmos que encriptan usando una clave especial para que sea ilegible para cualquiera que intercepte el mensaje.
En las computadoras modernas la longitud de la clave puede tener 100 bits y tardará aproximadamente $10^{18}$ veces más (un billón de billones) que una clave de 40 bits.

análisis
↑ teórico

## TABLA EXPERIMENTAL

Imaginemos $n^2$

| Tamaño n | $f(n)$ | $f(n)/n^2$ | $f(n)$/subestima | $f(n)$/sobreest. |
|---|---|---|---|---|
| 1000 | ~~Tiende a~~ | Tiende a cte | Oscila | tiende a cero |
| 10 000 | cte | | | |
| . | Valores | | | |
| . | | | | |
| . | | | | |
| 1000000 | | | | |

System.nanoTime()