

# Complejidad Algorítmica

🕒 Created	@March 3, 2022 7:03 PM
☰ Tags	2doTrimestre
▼ Formato	
▼ Materia	
▼ Tema	

## Big O

Es una métrica que mide la **eficiencia** de un algoritmo.

Cuando expresamos la complejidad en Big O, estamos indicando como el tamaño de un conjunto de datos afecta al rendimiento de un algoritmo.

## Algoritmo

Conjunto de instrucciones o reglas definidas y no-ambiguas, ordenadas y finitas que permite, típicamente, solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades.

## Tiempo de ejecución de un algoritmo

En BigO no podemos hablar de tiempo, ya que el tiempo dependerá de la máquina en la cual ejecutamos el algoritmo. Es una métrica que no depende de donde o quien ejecute el problema.

## Cantidad de información

No todos los algoritmos funcionan igual con la misma cantidad de información.

Algunos algoritmos funcionan bien cuando trabajan con poca cantidad de datos y mal con mucha cantidad de datos a tratar.

Y otros algoritmos al revés, son más eficientes cuanto tienen que trabajar con una gran cantidad de datos

## Tipos de complejidad

- **Complejidad en tiempo:** que tan eficiente es mi algoritmo para que tarde menos tiempo en realizar la tarea.
- **Complejidad en espacio:** cuanta memoria o espacio en disco usa mi algoritmo.

### ▼ Tipos de problema

- **Tratable:** Problema para el cual existe un algoritmo de complejidad polinomial. Se los conoce como problemas "P" (de orden polinomial).
- **Intratable:** Problema para el que no se conoce un algoritmo de complejidad polinomial. Se los conoce como problemas "NP" (de orden no determinístico polinomial).

---

## Tipos de Big O

### Notación Big O

$O(n)$

O = orden de la función

n = representa el tamaño de un conjunto

---

(Cuanto más bajemos peor es el algoritmo)

$O(1)$  → **Complejidad Constante:** La operación no depende del tamaño de los datos. Es el caso ideal, pero a la vez probablemente el menos frecuente.

$O(\log n)$  → **Complejidad Logarítmica:** Por regla general se asocia con algoritmos que "trocean" el problema para abordarlo, como por ejemplo una búsqueda binaria.

$O(n)$  → **Complejidad Lineal:** El tiempo de ejecución es directamente proporcional al tamaño de los datos. Crece en una línea recta. Cuanto más crecen los datos a tratar, más crece la complejidad.

$O(n * \log n)$  → **Complejidad Cuasi-lineal (logarítmica-lineal):** En este caso se trata de funciones similares a las anteriores, pero que rompen el problema en varios trozos por cada elemento, volviendo a recomponer información tras la ejecución de cada "trozo".

Por ejemplo, el algoritmo de orden Quicksort.

$O(n^2) \rightarrow$  **Complejidad Cuadrática**: Es típico de algoritmos que necesitan realizar una iteración por todos los elementos en cada uno de los elementos a procesar.

Por ejemplo el algoritmo de ordenación de burbuja. Si tuviese que hacer la iteración más de una vez serían de complejidad  $O(n^3)$ ,  $O(n^4)$ , etc... pero se trata de casos muy raros y poco optimizados.

Esta complejidad se da normalmente cuando usamos bucles for anidados.

$O(n^a) \rightarrow$  **Complejidad Polinomial ( $a \geq 2$ )**:

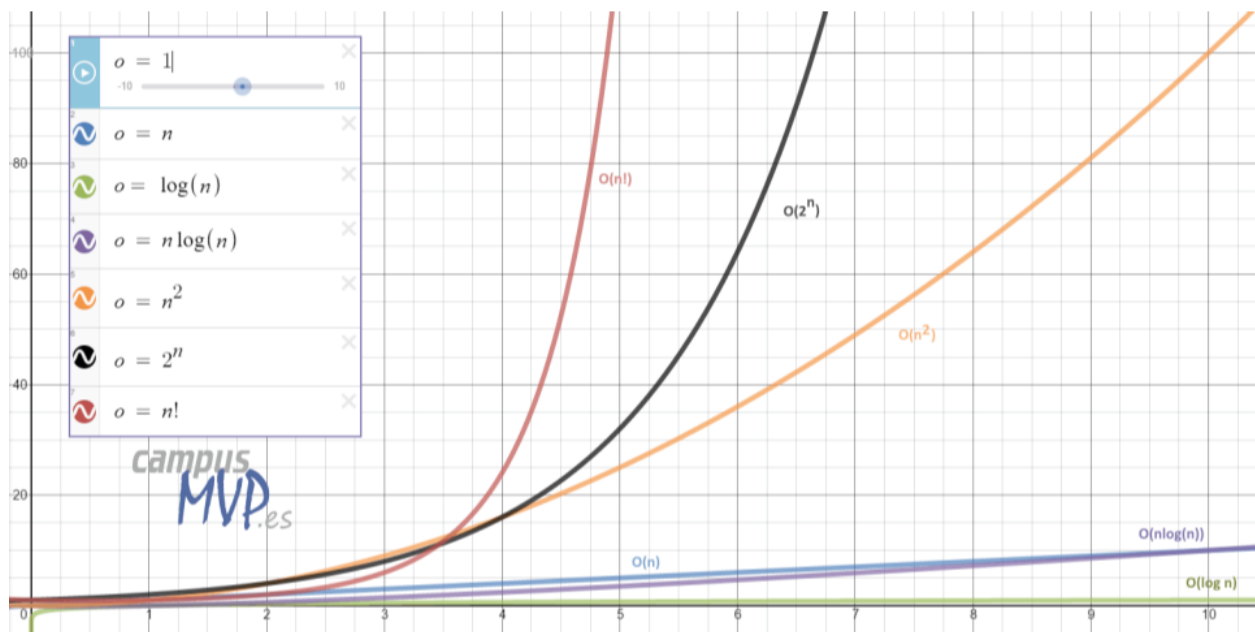
$O(a^n) \rightarrow$  **Complejidad Exponencial ( $a \geq 2$ )**: Se trata de funciones que duplican su complejidad con cada elemento añadido al procesamiento. Son algoritmos muy raros pues en condiciones normales no debería ser necesario hacer algo así.

Un ejemplo sería, por ejemplo, el cálculo recursivo de la serie de Fibonacci, que es muy poco eficiente (se calcula llamándose a sí misma la función con los dos números anteriores:  $F(n) = F(n - 1) + F(n - 2)$ ).

$O(n!)$   $\rightarrow$  **Complejidad Factorial = explosión combinatoria**: Un algoritmo que siga esta complejidad es un algoritmo totalmente fallido.

Una explosión combinatoria se dispara de tal manera que cuando el conjunto crece un poco, lo normal es que se considere computacionalmente inviable.

Solo se suele dar en algoritmos que tratan de resolver algo por la mera fuerza bruta. No deberías verlo nunca en un software "real".



## Términos no dominantes

Siempre vamos a simplificar la eficiencia descartando las constantes y los términos de orden inferior.

Ejemplos:

$$O(2n) = O(n)$$

$$O(n/2) = O(n * 1/2) = O(n)$$

$$O(4 + n) = O(n)$$

$$O(n/2 + n^2 + n^3) = O(n^3)$$

$$O(n + n^2) = O(n^2) \quad O(5 * 2^n + n^{100}) = O(2^n)$$

## ▼ ¿Por qué tener en cuenta esto?

- Si trabajamos con dispositivos con pocos recursos (baja memoria, poco procesador)
- Si trabajamos con grandes cantidades de datos
- Muy importante para sistemas en tiempo real
- Si queremos consumir menos energía (batería) o que el dispositivo se caliente menos.

- Si queremos que nuestra aplicación funcione más rápido

## Estructuras de datos más comunes

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

## Algoritmos de ordenación

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n^2)$	$\Theta(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n^2)$	$\Theta(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$\Theta(n(\log(n))^2)$	$\Theta(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$\Theta(n^2)$	$\Theta(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$\Theta(nk)$	$\Theta(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n)$