

big o



@heyimnowi

# big O

A la hora de escribir un algoritmo es importante tener en cuenta la complejidad y eficiencia del mismo.

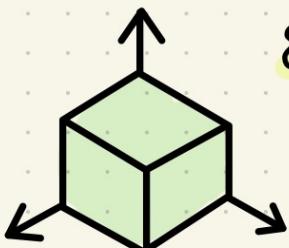
He aquí **BIG O**. ¿Qué es big O? Es una notación que permite determinar el rendimiento de un algoritmo. Nos permite describir la complejidad temporal y espacial del mismo.

## time complexity



La complejidad es una medida aproximada de la eficiencia. La complejidad temporal es una medida aproximada de cuánto tiempo le tomará a un algoritmo de procesar ciertas entradas

## space complexity



La complejidad espacial es una medida aproximada de cuánto espacio tomará un algoritmo para procesar ciertas entradas

mejor caso



peor caso

En algunos casos la complejidad de un algoritmo depende de los datos que tiene que procesar.

O big o

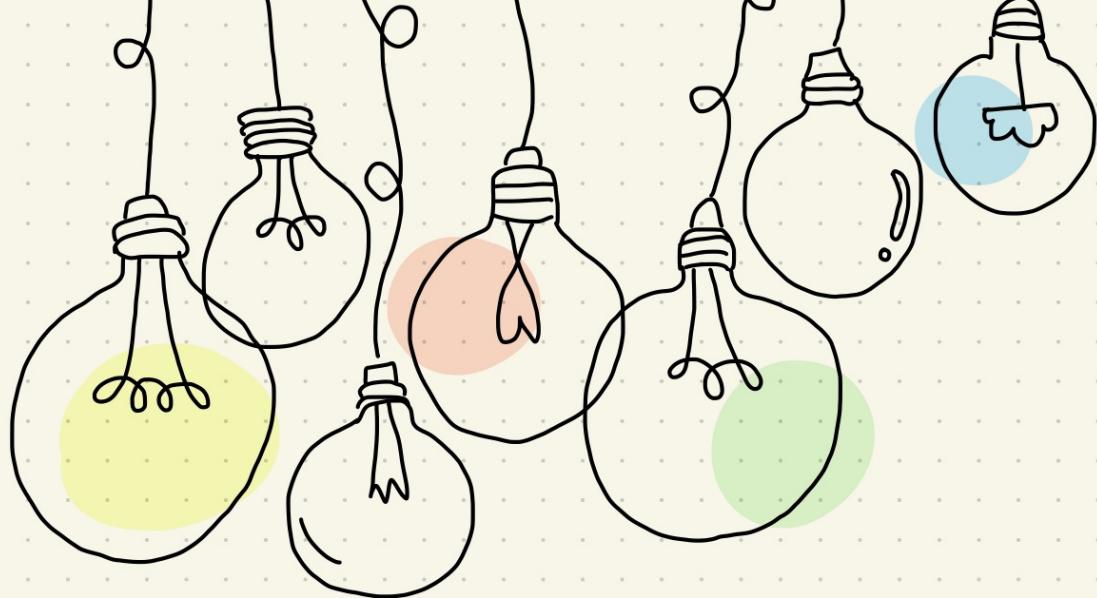
describe una cota superior del tiempo de ejecución.  
Aproximación para el peor caso.

$\Omega$  big omega

describe una cota inferior del tiempo de ejecución  
Aproximación para el mejor caso.

$\Theta$  big theta

describe una cota del tiempo de ejecución. Si un algoritmo es  $\Theta(N)$  significa que es  $O(N)$  y  $\Omega(N)$ . Aproximación para el caso promedio.



Como no siempre se espera que el algoritmo procese el mejor caso, generalmente la notación que más se usa es **big o**.

Supongamos que queremos ordenar una lista de números de forma ascendente usando el algoritmo **bubble sort**:



mejor caso

la lista ya está ordenada ascendente



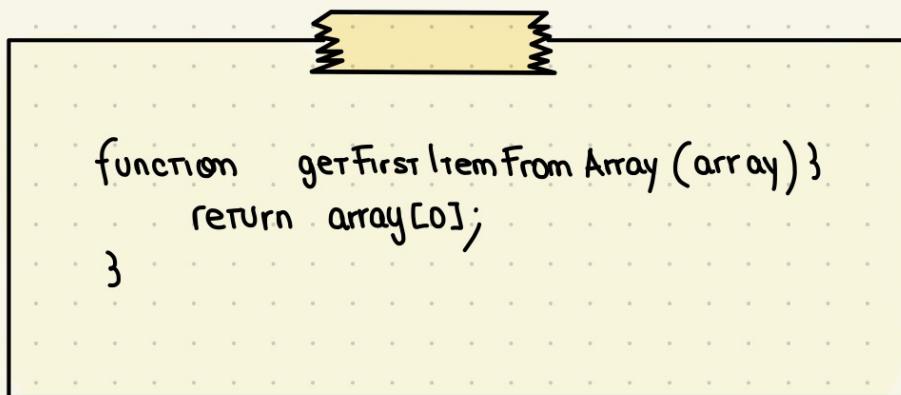
peor caso

la lista está ordenada descendente

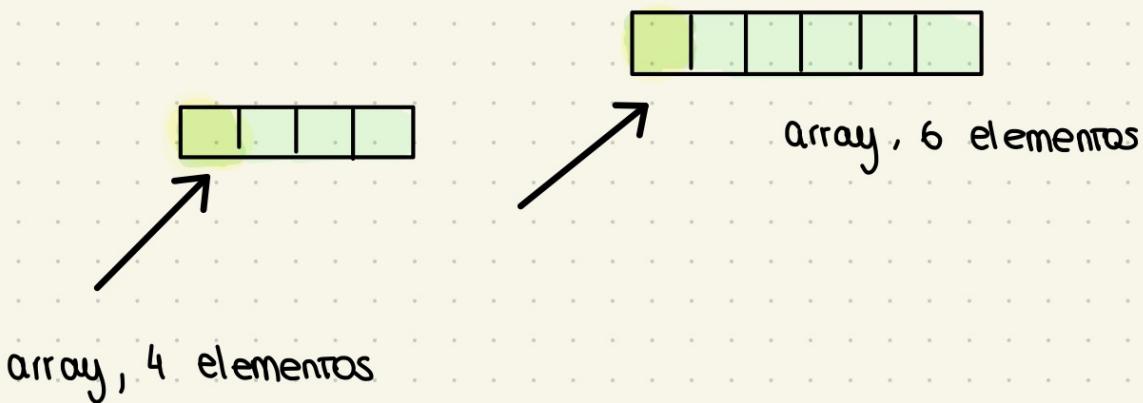
# principales órdenes

## O(1) constante

Este es el caso donde sin importar el tamaño de la entrada del algoritmo, el tiempo de ejecución y/o los recursos utilizados por el mismo son constantes.

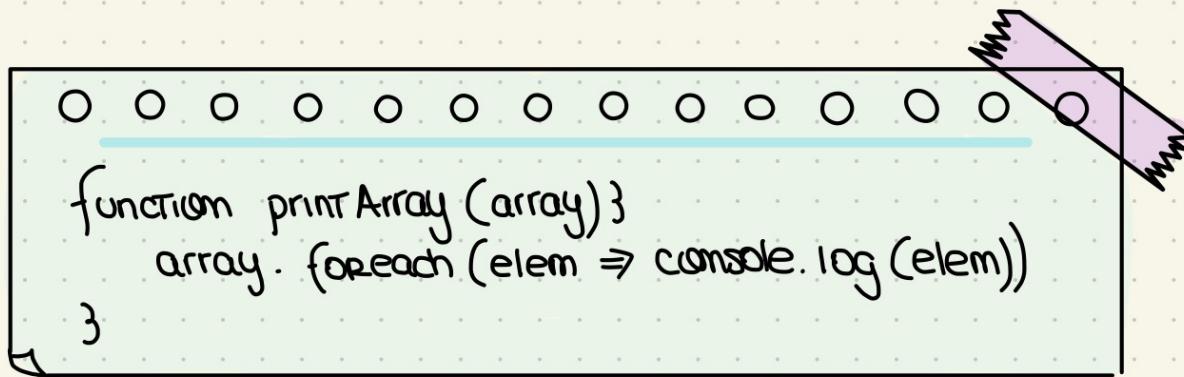


Un ejemplo de complejidad temporal constante es el obtener el primer elemento de un arreglo. No importa el tamaño del mismo, el resultado va a siempre igual. Lo que va a ir cambiando es el valor del primer elemento si las entradas no son las mismas.



# $O(n)$ linear

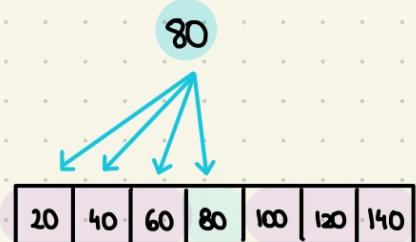
Cuando la complejidad de un algoritmo es linear es cuando depende proporcionalmente del Tamaño de la entrada.



Por ejemplo una función que imprime todos los elementos de un arreglo es de  $O(n)$  ya que hay que recorrer todos los elementos una vez y es directamente proporcional al tamaño del array.

## ejemplo

El algoritmo de búsqueda secuencial es un método para encontrar un valor en una lista comparándolos secuencialmente hasta encontrarlo o hasta que todos los elementos hayan sido comparados



mejor caso

Si el elemento a buscar es el primero de la lista

Complejidad:  $O(1)$



peor caso

Si el elemento a buscar es el último de la lista

Complejidad:  $O(n)$

# $O(\log n)$ logarítmica

La complejidad logarítmica es bastante común y es más fácil explicarla con el algoritmo de búsqueda binaria.

En una búsqueda binaria se busca un elemento  $X$  en un array de  $N$  elementos ordenado.

El algoritmo empieza comparando  $X$  con el elemento central del array, llamemoslo **middle**.

- ) Si  $X == \text{middle}$ , termina el algoritmo
- ) Si  $X < \text{middle}$ , se busca a  $X$  en el subarray izquierdo
- ) Si  $X > \text{middle}$ , se busca a  $X$  en el subarray derecho

Cuando veas un problema que donde el número de elementos se divide a la mitad en cada paso, es muy probable que sea  $O(\log n)$

# ejemplo

## BÚSQUEDA BINARIA

- 8 elementos
- elemento a buscar: 9

1	2	5	6	7	9	11	13
---	---	---	---	---	---	----	----

$$9 > 7$$

elijo el subarray derecho

9	11	13
---	----	----

$$9 < 11$$

elijo el subarray izquierdo

9
---

X es el elemento central y  
termina el algoritmo



# $O(2^n)$

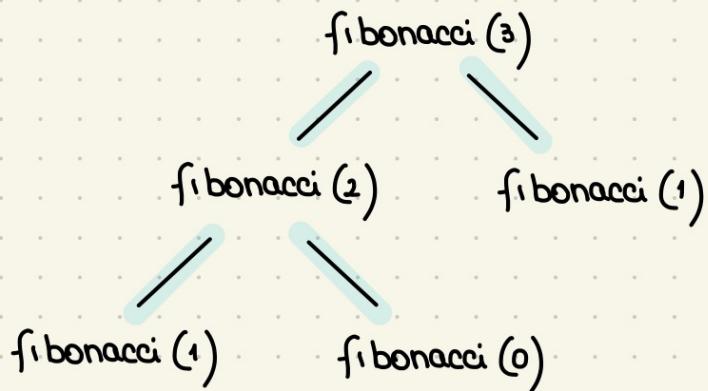
# exponencial

Es un orden que generalmente aparece en funciones recursivas. El típico ejemplo es la función para obtener la sucesión de Fibonacci.

```

fibonacci(n)
if (n <= 1) return 1;
return fibonacci(n-1) + fibonacci(n-2);
  
```

Supongamos que se llama a la función con  $n = 3$



Generalmente para funciones recursivas el orden es  $O(\text{branches}^{\text{depth}})$ . No aplica siempre

- **branches:** la cantidad de veces que en una llamada se llama a la función recursiva. En este caso es 2
- **depth:** es la cantidad de niveles del árbol de llamadas.

# $O(n^2)$

# cuadrática

Cuando la complejidad del algoritmo es proporcional al cuadrado del tamaño de la entrada se habla de  $O(n^2)$

Por ejemplo, si queremos verificar si un array tiene elementos repetidos habría que comparar uno a uno todos los elementos del array.

Ejemplos de algoritmos de complejidad cuadrática:

- Bubble sort
- Selection sort



# ejemplo



```
function printPairs (int [] array) {  
    for (int i = 0; i < array.length; i++) {  
        for (int j = 0; j < array.length; j++) {  
            console.log (array[i] + " , " + array[j]);  
        }  
    }  
}
```

3 3

Lo que hace la función es recorrer un array e imprimir sus elementos de a pares.

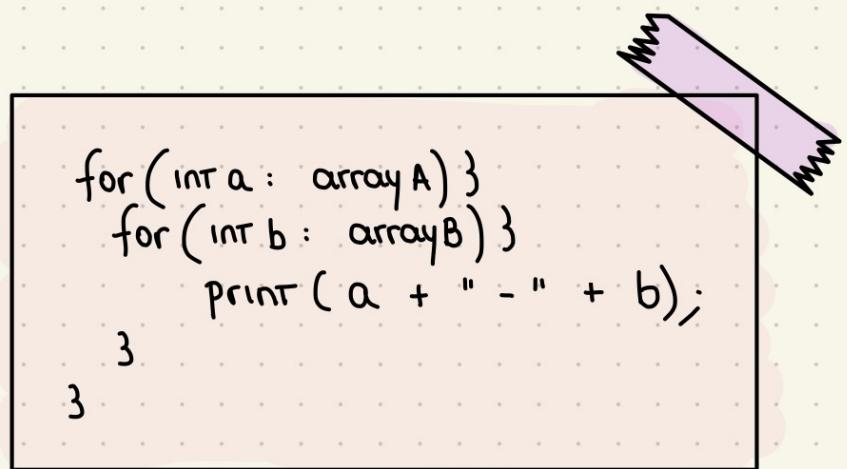
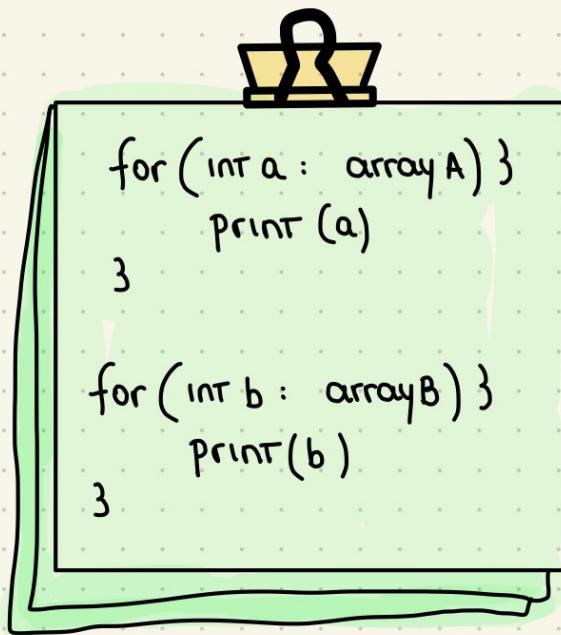
El primer loop se llama  $n$  veces y por lo tanto es  $O(n)$ .

El segundo loop tambien se llama  $n$  veces. Entonces la complejidad del algoritmo es  $O(n * n) = O(n^2)$



# $\Sigma$ sumar o multiplicar

Es muy común tener un algoritmo que consta de dos o más pasos. ¿Cuando se suman o se multiplican los tiempos de ejecución?



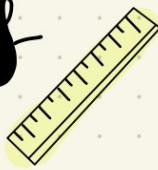
En el primer ejemplo se ejecuta el primer tramo una **a** cantidad de veces y el segundo tramo una cantidad **b** de veces.

Entonces el orden es  $O(a+b)$

En el segundo ejemplo, por cada elemento de **arrayA** se ejecuta la segunda línea una cantidad **b** de veces.

Entonces el orden es  $O(a * b)$

# reglas



Si el algoritmo es de la forma:

"hace esto y cuando termines hace esto OTRO"

Entonces los tiempos de ejecución se suman.

Si el algoritmo es de la forma:

"Hace esto por cada vez que hagas esto OTRO"

Entonces los tiempos de ejecución se multiplican

