Complejidad de algoritmos

AED I

2020

UBP UNIVERSIDAD BLAS PASCAL

Complejidad de Algoritmos

complejidad

- Para hacer un **uso** adecuado de los **recursos** computacionales (procesamiento, almacenamiento) es muy importante poder analizar la complejidad de los algoritmos

- Ese análisis nos permite darnos una idea de cómo el algoritmo se comportará en relación a la escala del conjunto de datos

- Responde a la pregunta: ¿cómo crecerá la cantidad de recursos necesarios en la medida en que se agreguen más y más datos a la entrada?

notación O

La notación O grande es la representación relativa de la complejidad de un algoritmo.

- También llamada notación asintótica o notación de Landau
- Describe la forma en que el tiempo de ejecución (o el espacio de almacenamiento) escalan con respecto al tamaño de la entrada

notación O

Si llamamos **n** al conjunto de datos

Podemos medir un algoritmo de acuerdo con su comportamiento en función al tamaño de **n** y esa medida será **f(n)**

No nos interesa **f(n)**, sino su tendencia en la medida que n aumenta **$n \rightarrow \infty$**

notación O

Sean algoritmos

Algoritmo 1 → $T(n) = 6\ n^2 + 9$

Algoritmo 2 → $T(n) = 14\ n^2 + 6\ n + 5$

Cuando n → ∞ las constantes y los términos de menor grado se hacen insignificantes y podemos caracterizar a ambos algoritmos con la misma tasa de crecimiento que es, en este caso, cuadrática

notación O

Definimos

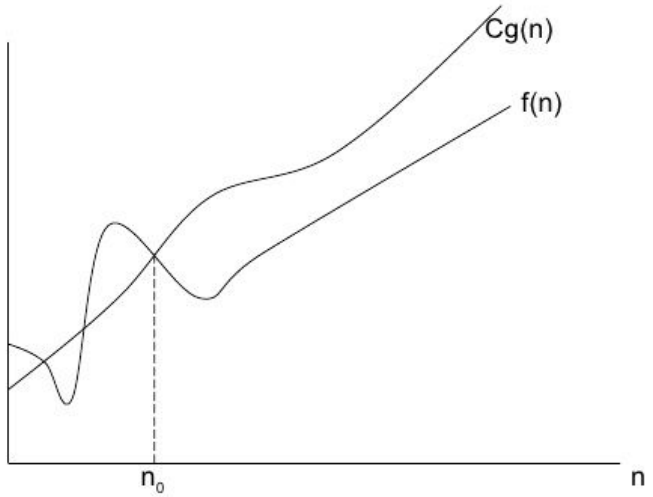
$$O(g(n)) = \{ f(n) : \exists\ c \in \mathbb{R} \ \wedge\ n_0 \in \mathbb{N} \ / \ f(n) \leq c \ . \ g(n) \text{ para } n \geq n_0 \}$$

Digamos que:

$f(n) = 5n^2 + 2n + 1 \in O(n^2)$

$g(n) = n^2$

$c = 8,\ f(n) \leq 8n^2\ ,\ n \geq 1$



f(n)=*O*(g(n))

notación Ω

Definimos

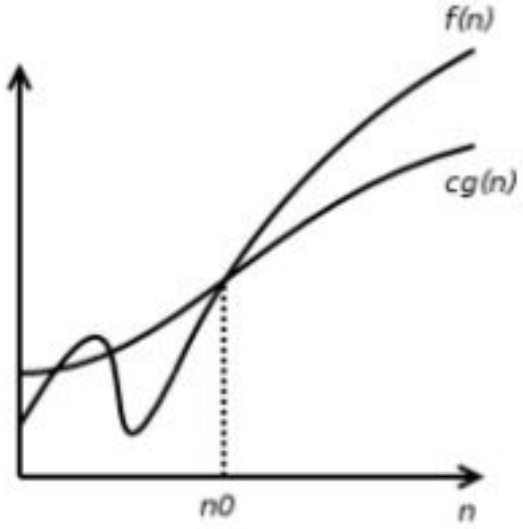
$$\Omega(g(n)) = \{ f(n) : \exists\ c \in \mathbb{R} \wedge n_0 \in \mathbb{N} / c \cdot g(n) \leq f(n) \text{ para } n \geq n_0 \}$$

Digamos que:

$f(n) = 5n^2 + 2n + 1 \in \boldsymbol{\Omega}(n^2)$

$g(n) = n^2$

$c = 5 \wedge n_0 = 0,\ 5n^2 \leq f(n),\ n \geq 0$



notación Θ

Definimos

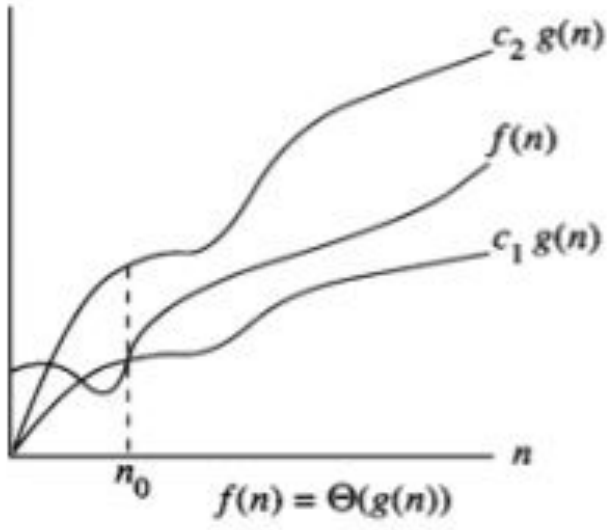
$\Theta(g(n)) = \{ f(n) : \exists\ c_1, c_2 \in \mathbb{R} \wedge n_0 \in \mathbb{N} / c_1 . g(n) \leq f(n) \leq c_2 . g(n) \text{ para } n \geq n_0 \}$

Digamos que:

$f(n) = 5n^2 + 2n + 1 \in \Theta(n^2)$

$g(n) = n^2$

$c_1 = 5, c_2 = 8, n_0 = 0,\ 5n^2 \leq f(n) \leq 8n^2, n \geq 0$



notación O: las más comunes

Constante	O(1)	`int i = a + b;`
Logarítmica	O(log n)	Búsqueda binaria

Lineal	O(n)	`for(i=0; i<n; i++){`
		`  ...`
		`}`

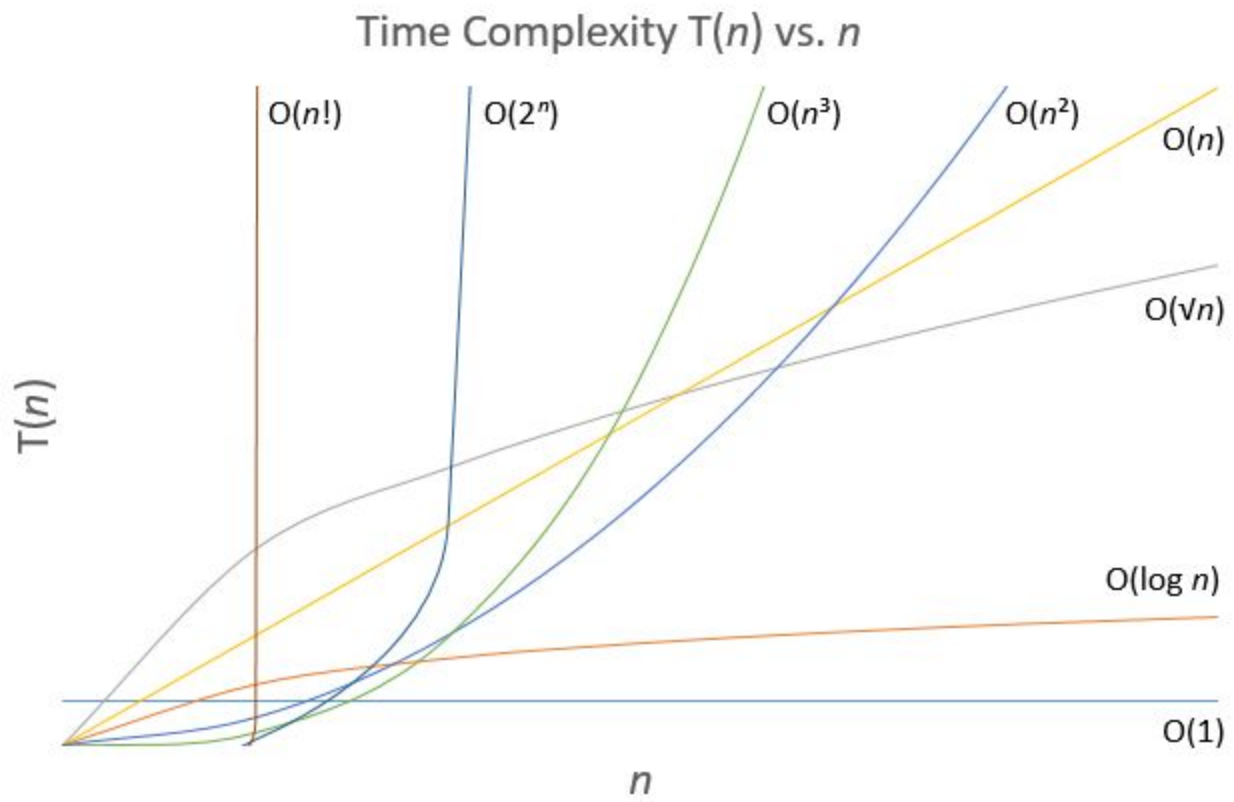
n log n	O(n log n)	Ordenamiento por Intercalación, Ordenamiento Rápido

Cuadrática	$O(n^2)$	`for(i=0; i<n; i++){`
		`  for(j=0; j<n; j++){`
		`  ...`
		`  }`
		`}`

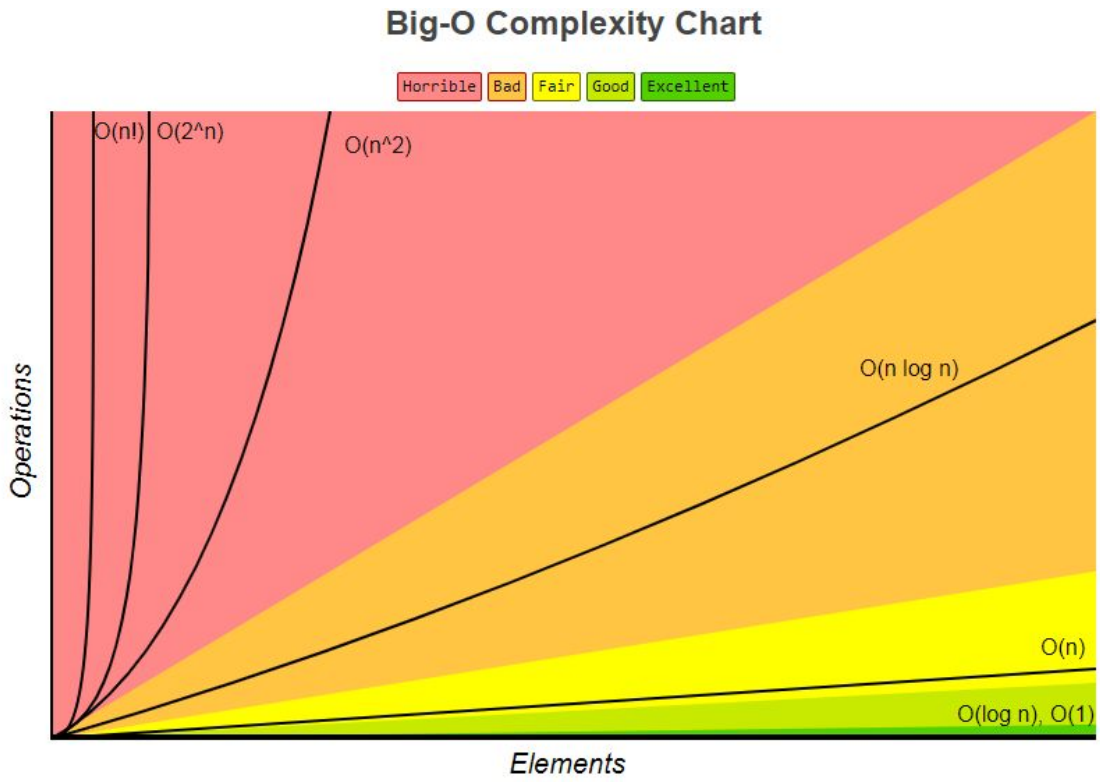
Cúbica	$O(n^3)$	`for(i=0; i<n; i++){`
		`  for(j=0; j<n; j++){`
		`  for(k=0; k<n; k++){`
		`  }`
		`  }`
		`}`

Exponencial	$O(2^n)$	Secuencia de Fibonacci recursiva
Factorial	O(n!)	Fuerza bruta

notación O



notación O



FUENTE: http://bigocheatsheet.com/

notación O

$$O(1) = O(😎)$$
$$O(\log(n)) = O(😁)$$
$$O((\log(n))^c) = O(😇)$$
$$O(n) = O(😍)$$
$$O(n \log(n)) = O(🙂)$$
$$O(n^{1.5}) = O(😐)$$
$$O(n^2) = O(☹)$$
$$O(n^c) = O(😣)$$
$$O(c^n) = O(😭)$$
$$O(n!) = O(😱)$$

FUENTE: https://github.com/corkami/docs/tree/master/BigEmo

O, Ω y Θ determinan relaciones entre funciones

. . . ∈ **O**(. . .) es una relación entre funciones:
"ser del O de" o "ser a lo sumo del orden de"
o "ser del orden de"

. . . ∈ **Ω**(. . .) es una relación entre funciones:
"ser del Ω de . . . " o "ser como mínimo del orden de"
o "ser del orden de"

. . . ∈ **Θ**(. . .) es una relación entre funciones:
"ser del Θ de . . . " o "ser exactamente del orden de"
o "ser del orden de"

notación O

Array Sorting Algorithms

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

FUENTE: http://bigocheatsheet.com/

notación O

Common Data Structure Operations

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

FUENTE: http://bigocheatsheet.com/

algunas reglas para su cálculo

1. Despreciar los términos de menor orden y las constantes multiplicativas

$T(n) = 3\,n^3 + 7\,n^2 + 6\,n + 5 = O(n^3)$

$T(n) = 12\,n + \log n = O(n)$

algunas reglas para su cálculo

2. Podemos calcular el tiempo de ejecución sumando el tiempo de ejecución de los fragmentos.

```
int a;
a = 5;
a++;
```

Fragmento 1

$O(1)$

```
for(int i=0; i<n; i++)
{
      /* hacer algo */
}
```

Fragmento 2

$O(n)$

```
for(int i=0; i<n; i++)
{
      for(int j=0; j<n; j++)
      {
            /* hacer algo */
      }
};
```

Fragmento 3

$O(n^2)$

algunas reglas para su cálculo

2. Podemos calcular el tiempo de ejecución sumando el tiempo de ejecución de los fragmentos.

```
int a;
a = 5;
a++;
```
O(1)

```
for(int i=0; i<n; i++)
{
      /* hacer algo */
}
```
O(n)

```
for(int i=0; i<n; i++)
{
      for(int j=0; j<n; j++)
      {
            /* hacer algo */
      }
};
```
$O(n^2)$

$$T(n) = O(1) + O(n) + O(n^2) = \mathbf{O(n^2)}$$