

Clase 8. Análisis de complejidad



TRABAJADORES
INFORMÁTICOS

organizados en AGC



Ministerio de Trabajo,
Empleo y Seguridad Social
Argentina

Cómo comparamos algoritmos?



Cómo comparamos algoritmos que tienen los mismos resultados? Como sabemos cual es mejor que otro? Que ideas tienen?

Ejemplo: Cual es mejor?

Python 

```
def buscar_maximo(numeros):  
    if (len(numeros) < 1):  
        return None  
    maximo = numeros[0]  
    for i in numeros:  
        if (i > maximo):  
            maximo = i  
    return maximo
```

Python 

```
def buscar_maximo(numeros):  
    if (len(numeros) < 1):  
        return None  
    for candidato in numeros:  
        comparados = 0  
        for i in numeros:  
            if (i > candidato):  
                break  
        comparados += 1  
    # Compare con todos, entonces encuentre el máximo  
    if (comparados == len(numeros)):  
        return candidato
```


Cómo comparamos algoritmos?

- Cuales ideas tenemos?


Cómo comparamos algoritmos?

- Cuales ideas tenemos?
- Medir tiempos? Ver cual termina antes? **NO**, por que puede variar de computadora en computadora.
- Contar operaciones? **NO**, por que la cantidad de operaciones suele variar dependiendo de la cantidad de datos.



Cómo comparamos algoritmos?

 Se llama eficiencia de un algoritmo a la forma de describir la cantidad de recursos utilizados para por un algoritmo

Vamos a usar dos tipos de complejidades:

- 
 - Temporal: Cuánto “tarda”
 - Espacial: Cuanto “ocupa”

Para qué sirve?

-  Para poder comparar algoritmos y elegir el más convenientes a nuestro caso de uso. Es un criterio que vamos a usar todos.
-  Usualmente es más útil para cuando la cantidad de datos es muy grande.

Operación elemental (complejidad temporal)

Vamos a definir como “operación elemental” a las operaciones básicas que vamos a decir que tienen complejidad **1**. Las operaciones elementales son:

- Asignación de variables.
- Chequeo de condicionales (**if**)
- Todas las operaciones elementales básicas.

Para calcular la complejidad, vamos a ver cuantas operaciones elementales hay.

Operación elemental (complejidad temporal)

Python 

```
def restar(a, b):  
    if (a == None):  
        return -b  
    if (b == None):  
        return a  
    return a - b
```

Para calcular la complejidad, se tiene en cuenta el “peor caso”, que es lo mismo que decir que vamos a tener en cuenta el camino más largo.

En este caso:

- `if (a == None):` 1
- `if (b == None):` 1
- `return a - b` 1

En total: **3** operaciones elementales.

Operación elemental (complejidad temporal)

Python 

```
def buscar_maximo(numeros):  
    if (len(numeros) < 1):  
        return None  
    maximo = numeros[0]  
    for i in numeros:  
        if (i > maximo):  
            maximo = i  
    return maximo
```

En este caso:

- `if (len(numeros) < 1):` 1
- `maximo = numeros[0]:` 1
- `for i in numeros:` La cantidad de veces depende de la longitud de `numeros`, si llamamos a dicha longitud **N**, entonces las operaciones de adentro del **for** se ejecutan N veces.
 - Dentro del **for** hay dos operaciones elementales: **2**.Entonces, todo el **for** queda en **N * 2**
- `return maximo:` 1

Total: $N * 2 + 3$

Operación elemental (complejidad temporal)

Python 

```
def funcion_confusa(numeros):  
    acum = 0  
    for i in numeros:  
        for j in numeros:  
            acum += j  
    return acum
```

En este caso:

- `acum = 0`: 1
- Luego vienen dos for. Si `numeros` tiene longitud **N**, el for interno se ejecuta N veces, y la línea dentro del mismo for se ejecuta N veces **por cada** iteración del for externo. Entonces todo ese bloque de código se ejecuta en $N * N = N^2$.
- `return acum`: 1

Total: $N^2 + 2$

Ejercicio

Python 

```
def funcion_confusa_7(a, b):  
    if (a < 10):  
        return b  
    r = 1  
    for i in range(a):  
        for j in range(b):  
            r = r * i * j  
    return r
```

Cual es la complejidad?

Notación O (O grande)

➤ La notación se compone como " $O(f(n))$ " donde $f(n)$ es una cota superior del funcionamiento del algoritmo. Lo que quiere decir, es que la complejidad del algoritmo es a lo sumo $f(n)$.

➤ Para calcular se utiliza el "peor caso". El caso donde el algoritmo "tarda mas" o "ocupa más espacio" (depende del tipo de complejidad que estemos analizando)

➤ Ejemplo: $O(n)$ de complejidad temporal quiere decir que en el peor caso el algoritmo es **lineal**, osea que recorre todos los elementos una cantidad fija de veces

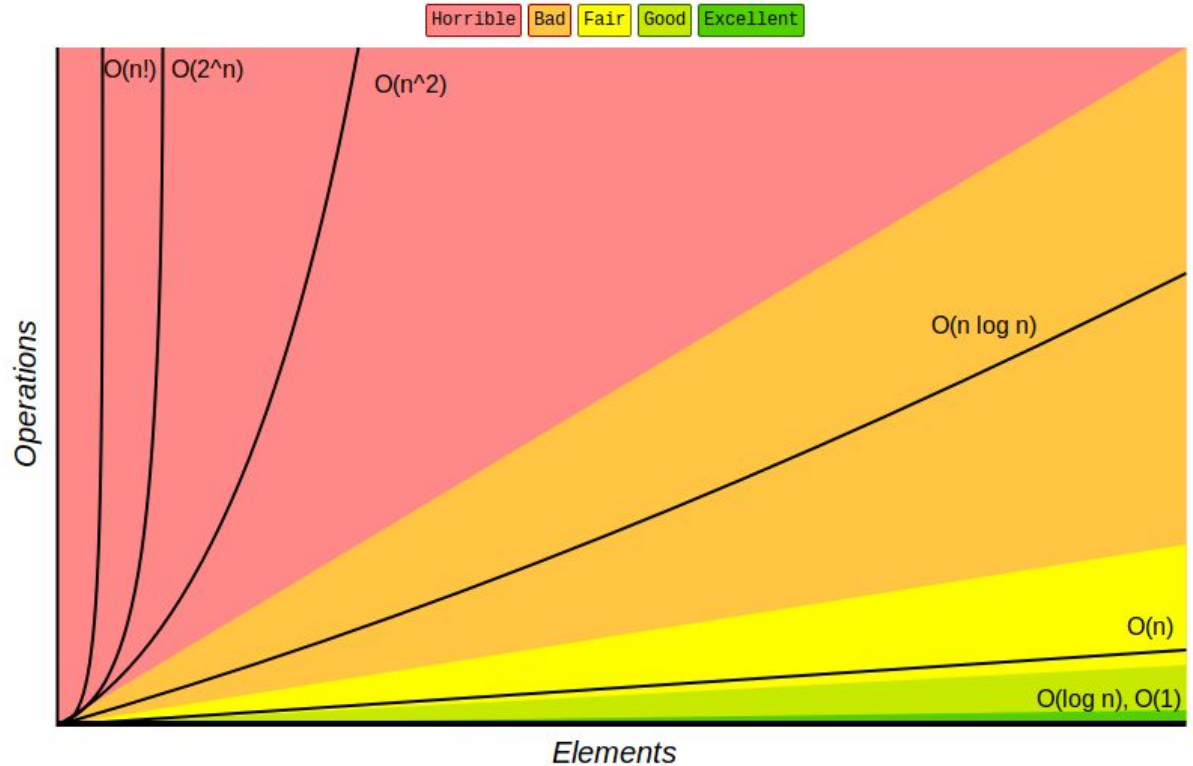
Notación O (O grande)

Notación	Nombre
$O(1)$	Orden constante. Siempre tarda lo mismo, no depende de los datos.
$O(\log n)$	Orden logarítmico.
$O(n)$	Orden lineal. Recorre todos los datos una cantidad fija de veces.
$O(n \cdot \log n)$	Orden lineal logarítmica
$O(n^2)$	Orden cuadrática. Recorre todos los datos una vez por cada dato.
$O(n!)$	orden factorial
$O(n^n)$	orden exponencial

Notación O (O grande)

Notación
$O(1)$
$O(\log n)$
$O(n)$
$O(n \cdot \log n)$
$O(n^2)$
$O(n!)$
$O(n^n)$

Big-O Complexity Chart



Operación elemental (complejidad temporal)

Para la notación O , vamos a ignorar las constantes y los multiplicadores y nos vamos a quedar con el elemento de la complejidad más grande (el que más nos impacta).

Esto es debido a que el factor determinante de la eficiencia del algoritmo es el orden más impactante y no las constantes. Por ejemplo $O(n^2)$ siempre va a ser peor que cualquier complejidad lineal, por mas que sea $O(1000 \cdot n)$, sigue siendo lineal. Se ve claramente cuando la cantidad de datos es grande.

Entonces las siguientes expresiones son equivalentes:

- $O(n^2 + 2n + 10) = O(n^2)$
- $O(n + 4) = O(n)$
- $O(n^3 + 7n^2 + 3n) = O(n^3)$
- $O(3) = O(1)$

Operación elemental (complejidad temporal)

Ejercicio: Reducir las siguientes complejidades:

- $O(17 + n)$
- $O(79 * n^2)$
- $O(n! + n^{20})$
- $O(n * \log(n) + n^2)$
- $O(n + \log(n) + 9)$

Ejercicio: Búsqueda simple

Python 

```
def buscar(vector, elemento):  
    index = 0  
    for i in vector:  
        if (i == elemento):  
            return index  
        index += 1  
    return -1
```

Cual es la complejidad temporal?

Ejercicio: Ordenamiento trivial

Python 

```
def buscar_maximo(numeros):
    if (len(numeros) < 1):
        return None
    pos = 0
    index = 0
    for candidato in numeros:
        if (candidato > numeros[pos]):
            pos = index
            index += 1
    return pos

def swap(vector, pos1, pos2):
    aux = vector[pos1]
    vector[pos1] = vector[pos2]
    vector[pos2] = aux

def ordenar(vector):
    for i in range(len(vector)):
        pos_maximo = buscar_maximo(vector[i:])
        swap(vector, i, i + pos_maximo)
    return vector
```

Cual es la complejidad temporal?

Preguntas teóricas

- Puede un algoritmo de búsqueda en un elemento en un vector **desordenado** tener una complejidad temporal menos a $O(n)$ siendo n la cantidad de elementos del vector?

Preguntas teóricas

- Puede un algoritmo de búsqueda en un elemento en un vector **desordenado** tener una complejidad temporal menos a $O(n)$ siendo n la cantidad de elementos del vector?

No! Por que si lo fuera, significaria que no es necesario revisar todos los elementos para ver si un elemento existe o no.

Preguntas teóricas

- Puede un algoritmo de búsqueda en un elemento en un vector **desordenado** tener una complejidad temporal menos a $O(n)$ siendo n la cantidad de elementos del vector?

No! Por que si lo fuera, significaria que no es necesario revisar todos los elementos para ver si un elemento existe o no.

- Siempre tenemos que buscar el algoritmo más **eficiente** o si la cantidad de datos es poca no es necesario?

Preguntas teóricas

- Puede un algoritmo de búsqueda en un elemento en un vector **desordenado** tener una complejidad temporal menos a $O(n)$ siendo n la cantidad de elementos del vector?

No! Por que si lo fuera, significaria que no es necesario revisar todos los elementos para ver si un elemento existe o no.

- Siempre tenemos que buscar el algoritmo más **eficiente** o si la cantidad de datos es poca no es necesario?

Si! Siempre hay que buscar la mejor solución, nunca se sabe cuando se va a necesitar escalar y la cantidad de datos puede aumentar.

Demostración en vivo: MergeSort ($n \cdot \log(n)$) vs selección (n^2)

MergeSort	Elementos	Tiempo (segundos)
	10	0,00002
	100	0,00039
	1000	0,00500
	10000	0,06400
Seleccion	10	0,00002
	100	0,00050
	1000	0,05000
	10000	6

[Mostrar código](#)

Complejidad espacial

- Se llama **complejidad espacial** al espacio en memoria que ocupa en términos de unidades elementales.
- Una variable es una unidad elemental.
- Un vector de N elementos, ocupa N unidades elementales.
- Ejemplos
 - Si la complejidad espacial es $O(1)$, complejidad constante, significa que ocupa una cantidad fija de espacio, no depende de los datos/
 - Si la complejidad es $O(n)$ siendo n la cantidad de datos, significa que ocupa tanto espacio como datos tenga.

Ejercicio: Búsqueda simple

Python 

```
def buscar(vector, elemento):  
    index = 0  
    for i in vector:  
        if (i == elemento):  
            return index  
        index += 1  
    return -1
```

Cual es la complejidad espacial?

Ejercicio: Búsqueda simple

Python 

```
def buscar(vector, elemento):  
    index = 0  
    for i in vector:  
        if (i == elemento):  
            return index  
        index += 1  
    return -1
```

Cual es la complejidad espacial?

$O(1)$ ya que solo almacena una variable (index). El vector no lo crea el algoritmo.

Ejercicio: Búsqueda simple

Python 

```
def cuadrado(vector):  
    vector_cuadrado = []  
    for i in vector:  
        vector_cuadrado.append(i * i)  
    return vector_cuadrado
```

Cual es la complejidad espacial?

Ejercicio: Búsqueda simple

Python 

```
def cuadrado(vector):  
    vector_cuadrado = []  
    for i in vector:  
        vector_cuadrado.append(i * i)  
    return vector_cuadrado
```

Cual es la complejidad espacial?

$O(n)$ siendo n la longitud del vector, ya que almacena un vector nuevo de tamaño n .