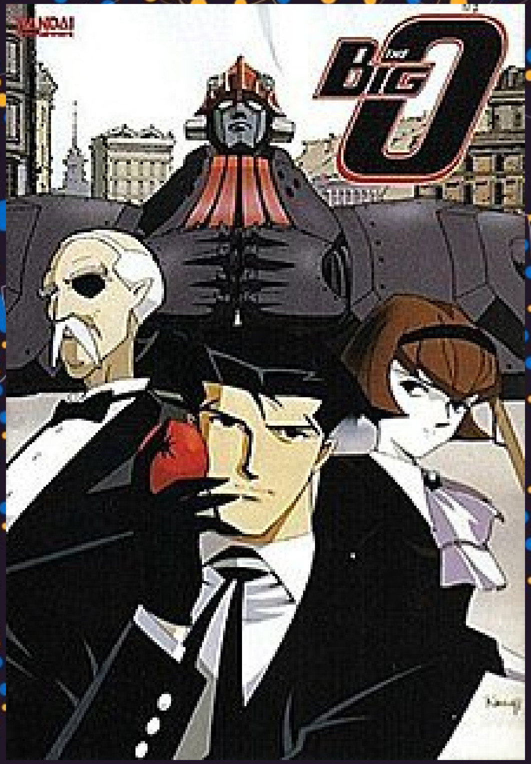
# *BIG O*

# Notación asintótica

Echo por: Cristian Israel Donato Flores



# *Antes de comenzar…*

→ ¿En qué consiste la eficiencia de un algoritmo?

- → Tiempo en ejecución
- → Cantidad de memoria utilizada
- → Cantidad de recursos computacionales consumidos durante la ejecución

→ ¿Cómo medir la calidad de un algoritmo?

- → En función de su eficiencia
- → Coste de escribirlo, entenderlo y modificarlo
- → Pero si se ejecuta pocas veces el peso relativo de escribirlo será importante y su eficiencia tendrá un menor valor

# *Antes de comenzar…*

→ Datos de entrada

→ En la mayoría de los casos los tiempos de ejecución están dados por los datos de entrada, es decir, dependen de su cantidad y no de su valor.

$$\rightarrow\ [2000,100,3000] < [0,1,2,3,4,5,6]$$

→ T(n)

→ Función del tiempo de ejecución de un algoritmo con *n* entradas.

→ Se expresa sin unidades.

→ Tal que n: número de operaciones elementales echas por un algoritmo.

# *Calculo por casos*

- → Ejecución mejor de los casos o caso mejor
  - → Es el número de operaciones elementales más favorables en la ejecución de un proceso.
    - → Obtenemos el menor tiempo de ejecución *T(n)*
    - → *Desventaja: Se obvian casi todos los posibles casos al ser demasiado "optimista"*
- → Ejecución promedio
  - → Utiliza la media de todos los tiempos de ejecución posible para un algoritmo y retorna una complejidad.
    - → *Inconveniente: No siempre se tiene la suficiente información para dicho calculo*

# *Calculo por casos*

- → Ejecución peor de los casos o caso peor
  - → Lo más recomendable es suponer el peor de los casos al calcular el tiempo de ejecución *T(n)*.
  - → *Ventaja: Se contemplan "todos" los casos posibles al suponer el peor como el tiempo de ejecución de T(n).*

**El calculo de *"ejecución en el peor de los casos"* es el que se implementa cuando utilizamos la notación *Big O*.**

# *Notaciones asintoticas*

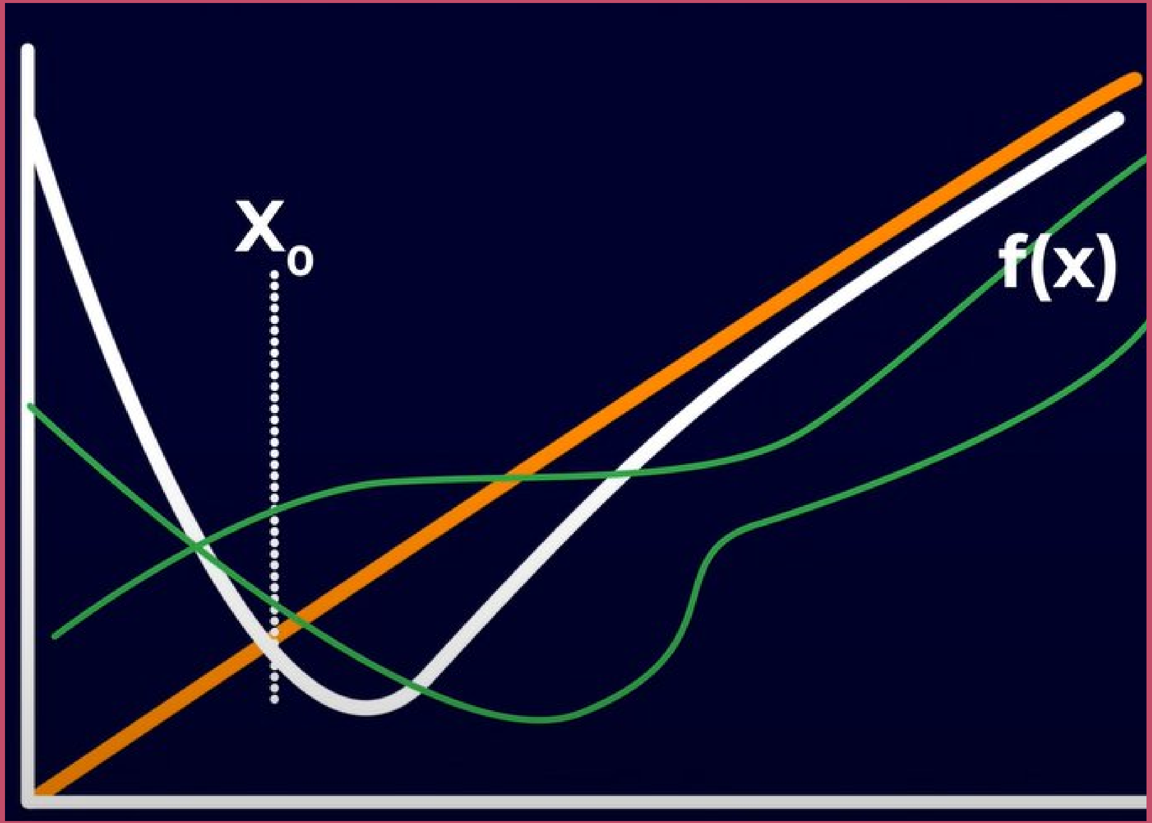
→ ¿Cuál es su fin?

→ Utilizada para describir el comportamiento de una función en base a su entrada

→ Es una forma de clasificar los algoritmos, existen 5 formas diferentes:

- Big O
- Big Ω
- Big θ
- Litte O
- Litte ω

# Big O

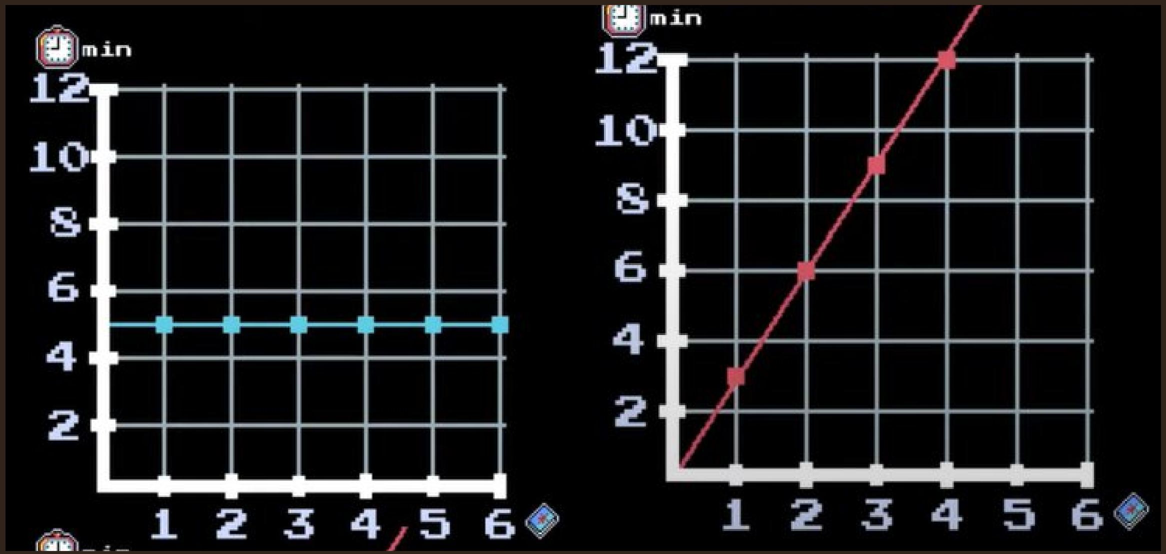



- → Define una cota superior o igual a la función del tiempo T(n)
- → A partir de un punto $x_0$ (también conocido como $n_0$) la función del tiempo siempre será menor o igual a nuestra cota superior
- → En palabras sencillas, definiremos una función que a partir de cierto punto siempre será superior o igual a la función del tiempo.

# *Análisis de algoritmos con Big O*

→ El análisis de los algoritmos es importante, ya que si nosotros basamos la eficiencia del tiempo de nuestro proceso midiendo el tiempo de ejecución pasa lo siguiente…

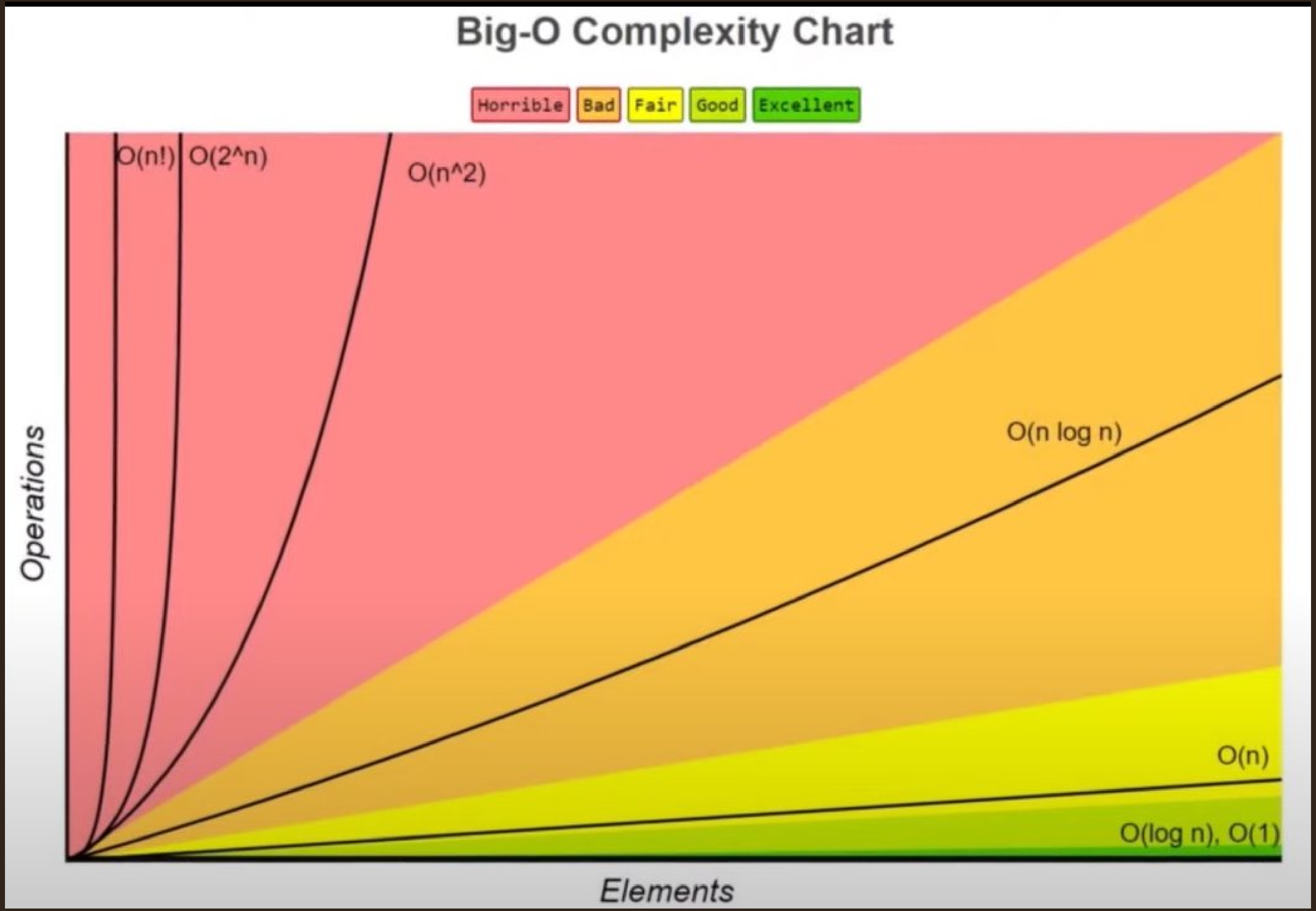
*Expectativa*




*Realidad*

# *Funciones y análisis de algoritmos con Big O*

| Velocidad de crecimiento | Nombre |
|---|---|
| **1** | constante |
| **log(n)** | logarítmica |
| **n** | lineal |
| **nlog(n)** | casi-lineal |
| **n^2** | cuadratica |
| **n^k** | polinómica |
| **2^n** | exponencial |
| **n!** | factorial |



# *Funciones de orden y análisis de algoritmos con Big O*

```
N = 1000
if N % 2 == 0:
    print "par"
else:
    print "impar"
```

O(1)

```
i = 1
while (i < N):
    print i
    i = i * 2
```

O(log N)

# Funciones de orden y análisis de algoritmos con Big O

```
for i = 0 to N:
    if i % 2 == 0:
        print i
```

O(N)

```
for i = 0 to N:
    if i % 2 == 0:
        print i
for i = 0 to N:
    if i % 2 != 0:
        print i
```

O(N)

```
for i = 0 to N:
    for j = i to N:
        print i + "," + j
```

$O(N^2)$

# *Funciones de orden y análisis de algoritmos con Big O*

```
for i = 0 to N:
    j = 1
    while j < N:
        print j
        j = j * 2
```

O(N*log N)

```
for i = 0 to length(A):
    for j = 0 to length(B):
        print A[i] + "," + B[j]
```

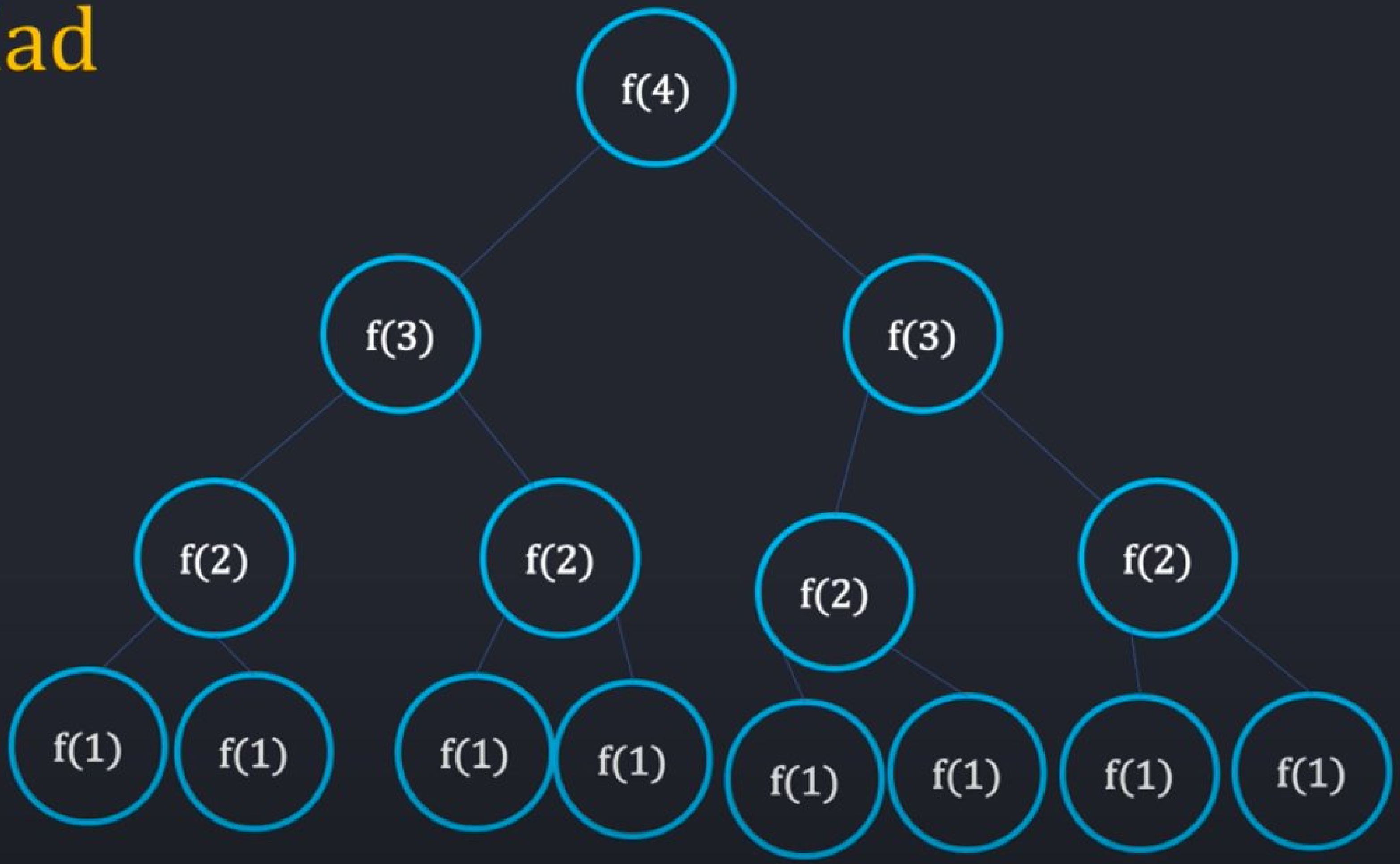
O(A*B)

# Funciones de orden y análisis de algoritmos con Big O

## Recursividad

```
int f(int n){
    if(n <= 1){
        return 1;
    }
    return f(n - 1) + f(n-1);
}
```

$O(2^N)$



# *Funciones de orden y análisis de algoritmos con Big O*

Después de haber echo el análisis de algoritmos con *Big O* debemos de sumar todas las complejidades  y simplificarlo.

```
public static void main(String []args){
    int []arr [redacted]                          // 1

    for(int j = 1; j < arr.length; j++){          // n
        int actual = arr[j];                      // n

        int i = j-1;                              // n
        while(i >= 0 && arr[i] > actual){         // n^2
            arr[i+1] = arr[i];                    // n^2
            i--;                                  // n^2
        }
        arr[i+1] = actual;                        // n
    }
}
```

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

# Ejercicios para clase

Determina el Big O de los siguientes códigos

→ Ejercicio

```
entrada = input()
x = 5

if entrada == "holi":
  print("saludo" * x)
```

```
// Donde "n" es la entrada
for (int i = 1; i <= n; i += c) {
  // Cualquier sentencia O(1)
}
```

→ Solución

```
entrada = input()            # O(1)
x = 5                        # O(1)

if entrada == "holi":        # O(1)
  print("saludo" * x)        # O(1)
```

$\therefore O(1)$

```
// Donde "n" es la entrada
for (int i = 1; i <= n; i += c) { // O(n)
  // Cualquier sentencia O(1)
}
```

$\therefore O(n)$

# Ejercicios para clase

Determina el Big O de los siguientes códigos

→ Ejercicio

```
// Donde "c" NO varía con la entrada
for (int i = 1; i <= c; i++) {
  // Cualquier sentencia O(1)
}
```

→ Solución

```
// Donde "c" NO varía con la entrada
for (int i = 1; i <= c; i++) {  // O(1)
  // Cualquier sentencia O(1)
}
```

$\therefore O(1)$

```
// Donde "n" es la entrada
for (int i = 1; i <=n; i += c) {
  for (int j = 1; j <=n; j += c) {
    // Cualquier sentencia O(1)
  }
}
```

```
// Donde "n" es la entrada
for (int i = 1; i <=n; i += c) {
  for (int j = 1; j <=n; j += c) { // O(n^2)
    // Cualquier sentencia O(1)
  }
}
```

$\therefore O(n\text{^}2)$

# *Ejercicios para clase*

Determina el Big O de los siguientes códigos

→ Ejercicio

```
// Donde "n" es la entrada
for (int i = 1; i <=n; i *= c) { /
  // Cualquier sentencia O(1)
}
```

```
// Donde "n" es la entrada
for (int i = 2; i <=n; i = pow(i, c)) {
  // Cualquier sentencia O(1)
}
```

→ Solución

```
// Donde "n" es la entrada
for (int i = 1; i <=n; i *= c) { // O(log[n])
  // Cualquier sentencia O(1)
}
```

$\therefore O(\log n)$

```
// Donde "n" es la entrada
for (int i = 2; i <=n; i = pow(i, c)) { // O(log[log[n]])
  // Cualquier sentencia O(1)
}
```

$\therefore O(\log[\log[n]])$

# Ejercicios para clase

Determina el Big O de los siguientes códigos

→ Ejercicio

```java
public static void main(String []args){
    int []arr = {5, 3, 4, 8, 7, 5, 1, 2, 3};

    for(int j = 1; j < arr.length; j++){
        int actual = arr[j];

        int i = j-1;
        while(i >= 0 && arr[i] > actual){
            arr[i+1] = arr[i];
            i--;
        }
        arr[i+1] = actual;
    }
}
```

→ Solución

```java
public static void main(String []args){
    int []arr = {5, 3, 4, 8, 7, 5, 1, 2, 3};      // 1

    for(int j = 1; j < arr.length; j++){          // n
        int actual = arr[j];                      // n

        int i = j-1;                              // n
        while(i >= 0 && arr[i] > actual){         // n²
            arr[i+1] = arr[i];                    // n²
            i--;                                  // n²
        }
        arr[i+1] = actual;                        // n
    }
}
```

Ya que *n* es constante $\therefore O(1)$

# *Referencias*

→ https://youtu.be/MyAiCtuhiqQ

→ https://youtu.be/IZgOEC0NIbw