

# Ejercitación: Análisis de Complejidad por casos

Mauro José Sorbello Diaz

Legajo: 14006

## Ejercicio 1:

Demuestre que  $6n^3 \neq O(n^2)$ .

Ejercicio 1:

$$\lim_{n \rightarrow \infty} \frac{n^2}{6n^3} = \lim_{n \rightarrow \infty} \frac{1}{6n} = \frac{1}{6} \lim_{n \rightarrow \infty} \frac{1}{n} = \infty \text{ (no)} \quad \text{Entonces } n^3 \neq O(n^2)$$

## Ejercicio 2:

¿Cómo sería un array de números (mínimo 10 elementos) para el mejor caso de la estrategia de ordenación Quicksort(n) ?

Una lista de números para el mejor caso de Quicksort, ocurre cuando están equilibrados por igual los dos lados de la partición a cada nivel de la recursión. Osea el pivote produce dos sublistas que contienen el mismo número de elem.

Ej:  $A = [1, 7, 12, 9, 3, 5, 10, 4, 2, 11, 8, 6]$

## Ejercicio 3:

Cuál es el tiempo de ejecución de la estrategia **Quicksort(A)**, **Insertion-Sort(A)** y **Merge-Sort(A)** cuando todos los elementos del array A tienen el mismo valor?

En Quicksort(A), dependerá de donde se elija el pivot, si el pivot se encuentra justo en el medio, será  $O(n\log n)$ . Si el pivot es al azar, también será  $O(n\log n)$

En Insertion-Sort(A): Al no estar ordenados, porque son todos iguales.  $n$  elementos se comparan  $n$  veces, es decir,  $n * n = n^2 = O(n^2)$

En Merge-Sort(A) la complejidad será también de  $O(n\log n)$  dado a que se da en el peor de los casos.

## Ejercicio 4:

Implementar un algoritmo que ordene una lista de elementos donde siempre el elemento del medio de la lista contiene antes que él en la lista la mitad de los elementos menores que él. Explique la estrategia de ordenación utilizada.

### Ejemplo de lista de salida

7	3	2	8	5	4	1	6	10	9
---	---	---	---	---	---	---	---	----	---

```
33 v def ordenar(L):
34     #ORDENO LA LISTA
35     QuickSort(L)
36     mitad = int(length(L)/2)
37     print(mitad)
38     current = L.head
39 v     for i in range(0, mitad - 1):
40         current = current.nextNode
41         print(current.value)
42     start = L.head
43 v     for i in range(0, int((mitad -1) /2)):
44         vals = start.value
45         start.value = current.nextNode.value
46         current.nextNode.value = vals
47         current = current.nextNode
48         start = start.nextNode
49     print("LISTA A")
50     val(A)
51 ordenar(A)
```

Primero ordene la lista con un Quicksort.

Después busca el valor del medio de la lista.

Por último divide la mitad de la longitud de la lista en 2 partes. E intercambia la mitad de los nodos menores al del medio por mayores al del medio.

## Ejercicio 5:

Implementar un algoritmo **Contiene-Suma(A,n)** que recibe una lista de enteros A y un entero n y devuelve True si existen en A un par de elementos que sumados den n. Analice el costo computacional.

```
5  def ContieneSumaAux(A, n, current):
6      if current != None:
7          newCurrent = current
8          check = False
9          while check == False:
10             suma = current.value + newCurrent.value
11             if suma == n:
12                 return True
13             else:
14                 if newCurrent.nextNode != None:
15                     newCurrent = newCurrent.nextNode
16                 else:
17                     check = True
18     if current.nextNode != None:
19         return ContieneSumaAux(A, n, current.nextNode)
20
21 def ContieneSuma(A, n):
22     if A.head != None:
23         return ContieneSumaAux(A, n, A.head)
```

La complejidad del algoritmo es  $O(n^2)$ . Dado a la recursividad y el while loop.

## Ejercicio 6:

Investigar otro algoritmo de ordenamiento como BucketSort, HeapSort o RadixSort, brindando un ejemplo que explique su funcionamiento en un caso promedio. Mencionar su orden y explicar sus casos promedio, mejor y peor.

### BucketSort

Bucket sort es un algoritmo de ordenación que funciona distribuyendo los elementos de una matriz en varios buckets (ARRAYS). A continuación, cada bucket se ordena individualmente, ya sea utilizando un algoritmo de ordenación diferente o aplicando recursivamente el algoritmo de ordenación del bucket.

La ordenación de cubos es útil principalmente cuando la entrada se distribuye uniformemente en un rango.

Cada Bucket tiene una condición excluyente para poder ingresar un elemento dentro.

### Ejemplo caso promedio:

Supongamos que el Array es:

A = [0.15, 0.27, 0.11, 0.35, 0.22, 0.21]

- 1) Entonces creamos un Array con length 6:

B = [0, 0, 0, 0, 0, 0]

Y en el B(0) solo entraran los numeros del 0 al 0.1 (sin incluir al 0.1), B(1) los numeros 0.1 al 0.2, en el B(2) los numeros del 0.2 al 0.3 (sin incluir el 0.2), etc.

- 2) B=[0, [0.15, 0.11], [0.27, 0.22, 0.21], 0.35, 0, 0]
- 3) Luego: Utilizamos algun algoritmo de ordenamiento para ordenar los elementos de cada Bucket no vacio.
- 4) Por ultimo insertamos en una nueva matriz, los elementos ordenados de cada bucket no vacio.

C = [0.11, 0.15, 0.21, 0.22, 0.27, 0.35]

## COMPLEJIDAD:

### Peor caso:

Cuando la mayor cantidad de elementos se colocan en el mismo cubo. Esto puede dar lugar a que algunos cubos tengan más elementos que otros.

Esto hace que la complejidad dependa del algoritmo de ordenamiento que utilicemos.

Por ejemplo si utilizamos el InsertionSort, la complejidad sera de O( $n^2$ )

### Caso promedio:

Cuando los elementos se distribuyen aleatoriamente en la matriz. Incluso si los elementos no se distribuyen uniformemente, la ordenación del bucket se ejecuta en tiempo lineal. Es válido hasta que la suma de los cuadrados de los tamaños de cubo sea lineal en el número total de elementos.O(n)

### Mejor caso:

Cuando los elementos se distribuyen uniformemente en los cubos con un número casi igual de elementos en cada cubo.

La complejidad se vuelve aún mejor si los elementos dentro de los cubos ya están ordenados.

La complejidad de tiempo de la ordenación de cubos es O( $n + k$ ), donde n es el número de elementos y k es el número de cubos.

## Ejercicio 7:

A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en  $\Theta(n)$  y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para  $n \leq 2$ . Resolver 3 de ellas con el método maestro completo:  $T(n) = a T(n/b) + f(n)$  y otros 3 con el método maestro simplificado:  $T(n) = a T(n/b) + n^c$

- a.  $T(n) = 2T(n/2) + n^4$
- b.  $T(n) = 2T(7n/10) + n$
- c.  $T(n) = 16T(n/4) + n^2$
- d.  $T(n) = 7T(n/3) + n^2$

$$e. T(n) = 7T(n/2) + n^2$$

$$f. T(n) = 2T(n/4) + \sqrt{n}$$

$$f) a) 2T\left(\frac{n}{2}\right) + n^4$$

$$a=2, b=2, c=4$$

$$\log_2(2) = 1$$

$$\text{Caso 3: } 1 < 4, T(n) = \Theta(f(n)) = \Theta(n^4)$$

$$b) 2T\left(\frac{n}{10}\right) + n$$

$$a=2, b=\frac{10}{4}, f(n) = n$$

$$\log_{\frac{10}{4}}(2) = 1,94$$

$$\text{Caso 1: } n^{\log_{\frac{10}{4}}(2)-\epsilon} = f(n) \text{ cuando } \epsilon = 0,94$$

~~Observación:~~ Por lo tanto, por el Caso 1, nuestro  $T(n) = \Theta(n^{\log_{\frac{10}{4}}(2)}) = \Theta(n^{1,94})$

$$c) T(n) = 16T\left(\frac{n}{4}\right) + n^2$$

$$a=16, b=4, f(n) = n^2$$

$$\log_4(16) = 2$$

$$\text{Caso 2: Tenemos que } f(n) = \Theta\left(n^{\log_4(16)}\right) = \Theta(n^2)$$

Por lo tanto para el caso 2 nuestro  $T(n) = \Theta(n^2 \log n)$

$$d) T(n) = 7T\left(\frac{n}{3}\right) + n^2$$

$$a=7, b=3, c=2$$

$$\log_3(7) = 1,77$$

Como  $\log_3(7) < 2$ ,  $T(n) = \Theta(f(n)) = \Theta(n^2)$  Por caso 2

$$e) T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

$$a=7, b=2, c=1$$

$$\log_2(7) = 2,8$$

Como  $\log_2(7) > 2$ ,  $T(n) = \Theta(n^{2,8})$

$$7) f) T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$$

$$d=2, b=4, f(n)=n^{\frac{1}{2}}$$

$$\log_4(2) = \frac{1}{2}$$

$$\text{Por caso dos: Tenemos que } f(n) = \Theta\left(n^{\log_4(2)}\right) = \Theta\left(n^{\frac{1}{2}}\right)$$

$$\text{Por lo tanto, nuestro } T(n) = \Theta\left(n^{\frac{1}{2}} \lg n\right)$$