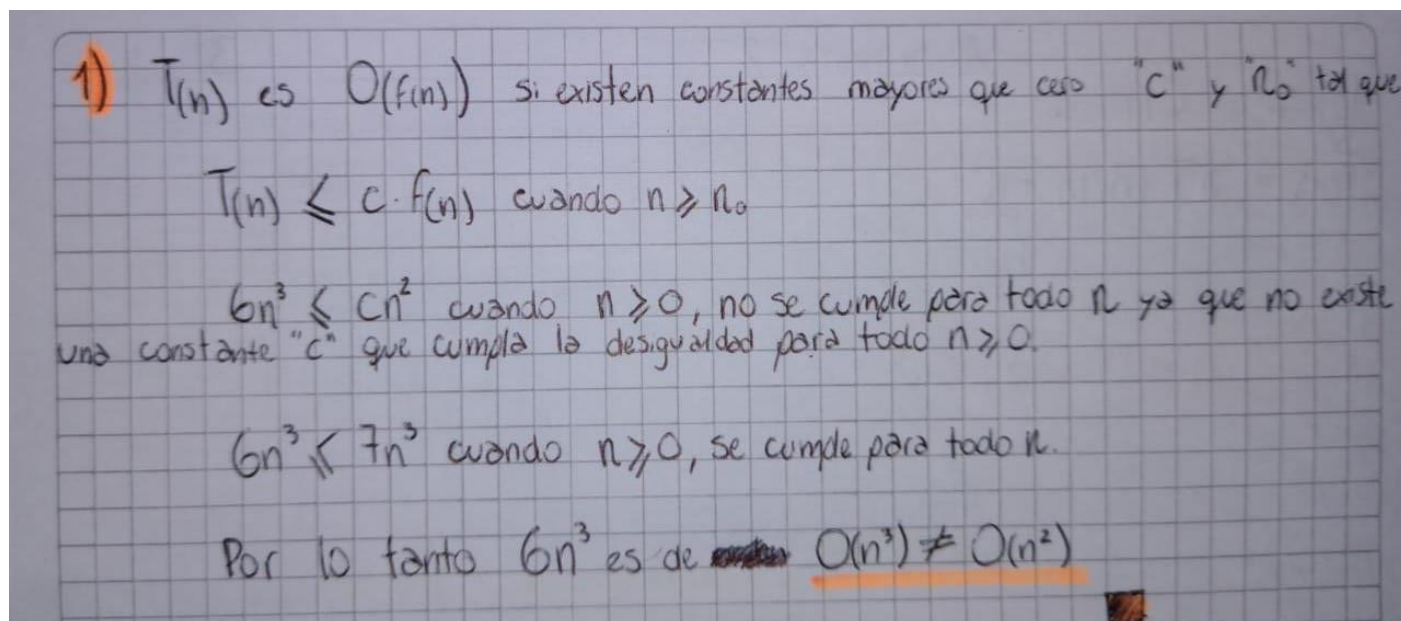


NAHMAN MARTINA L:13685

Ejercicio 1:

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



Ejercicio 2:

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

En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es **$O(n \cdot \log n)$** .

Array:

6	2	4	1	5	9	15	10	13	32	11
---	---	---	---	---	---	----	----	----	----	----

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?

- QuickSort(A): $O(n^2)$
- Insertion-Sort(A): $O(n)$
- Merge-Sort(A): $O(n \log n)$

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 entrada

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

```
#Ejercicio 4
def order_list(List):
    #Encuentro el elemento del medio
    pivot = List[len(List)//2]
    menores = []
    mayores = []
    #Recorremos lista y llenamos las otras
    for i in range(len(List)):
        if i == pivot:
            continue
        if List[i] < List[pivot]:
            menores.append(List[i])
        else:
            mayores.append(List[i])
    menores_new = orderList(menores)
    mayores_new = order_list(mayores)
    # + concatena las listas y forma la nueva lista
    return menores_new + [List[pivot]] + mayores_new
```

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.

```
#Ejercicio 5
def ContieneSuma(A, n):
    current = A.head
    aux = A.head
    if aux.value+current.nextNode.value==n:
        return True
    else:
        while current != None and aux != None: #o(n) peor caso-#O(C)
            if current.nextnode != None:
                if aux.value + current.nextnode.value == n:
                    return True
                current = current.nextNode
            else:
                aux = aux.nextNode
                current = aux
        return
```

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.

Counting sort:

El ordenamiento por cuentas (counting sort en inglés) es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Solo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [*mínimo*, *máximo*], y a cada

elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

Se trata de un algoritmo estable cuya complejidad computacional es $O(n+k)$, siendo n el número de elementos a ordenar y k el tamaño del vector auxiliar (*máximo - mínimo*).

La eficiencia del algoritmo es independiente de lo casi ordenado que estuviera anteriormente. Es decir no existe un mejor y peor caso, todos los casos se tratan iguales.

El algoritmo counting, no se ordena **in situ**, sino que requiere de una memoria adicional.

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. $T(n) = 2T(n/2) + n^4$ (6)
- b. $T(n) = 2T(7n/10) + n$ (2)
- c. $T(n) = 16T(n/4) + n^2$ (5)
- d. $T(n) = 7T(n/3) + n^2$ (4)
- e. $T(n) = 7T(n/2) + n^2$ (3)
- f. $T(n) = 2T(n/4) + \sqrt{n}$ (1)

Handwritten solution for exercise 7a using the Master Theorem. The recurrence is $T(n) = 2T(n/2) + n^4$. The parameters are identified as $a=2$ and $b=2$. The work per level is $f(n) = n^4$. The critical exponent is calculated as $n^{\log_2(2)} = n^1$. The solution is identified as Case 3, where $f(n) = O(n^{1+\epsilon})$ with $\epsilon = 3$. The final complexity is $\Theta(n^4)$.

b) $T(n) = 2T(n/10) + n$

$a = 2$

$b = \frac{10}{7}$

$c = 1$

$\log_{10/7} 2 = 1,94$

$\log_b a > c \rightarrow \text{caso 1}$

$\Theta(n^{\log_b(a)})$

c) $T(n) = 16T(n/4) + n^2$

$a = 16$

$b = 4$

$c = 2$

$\log_4 16 = 2$

$\log_b a = c \rightarrow \text{caso 2}$

$\Theta(f(n) \log n) = \Theta(n^c \log n)$

d) $T(n) = 7T(n/3) + n^2$

$a = 7$

$b = 3$

$f(n) = n^2$

$n^{\log_3(7)} = n^{1,77}$

$\text{caso 3 } f(n) = O(n^{1,77+\epsilon})$

$\epsilon \approx 0,23$

e) $T(n) = 7T(n/2) + n^2$
 $a=7$ $n^{\log_2(7)} = n^{2,81}$
 $b=2$
 $f(n) = n^2$
caso 1 $f(n) = O(n^{2,81-\epsilon})$ $\epsilon \approx 0,81$

f) $T(n) = 2T(n/4) + \sqrt{n}$
 $a=2$ $\log_4 2 = \frac{1}{2}$
 $b=4$ $\log_b a = c \rightarrow$ caso 2
 $c = \frac{1}{2}$
 $\Theta(f(n) \lg n) = \Theta(n^c \lg n)$