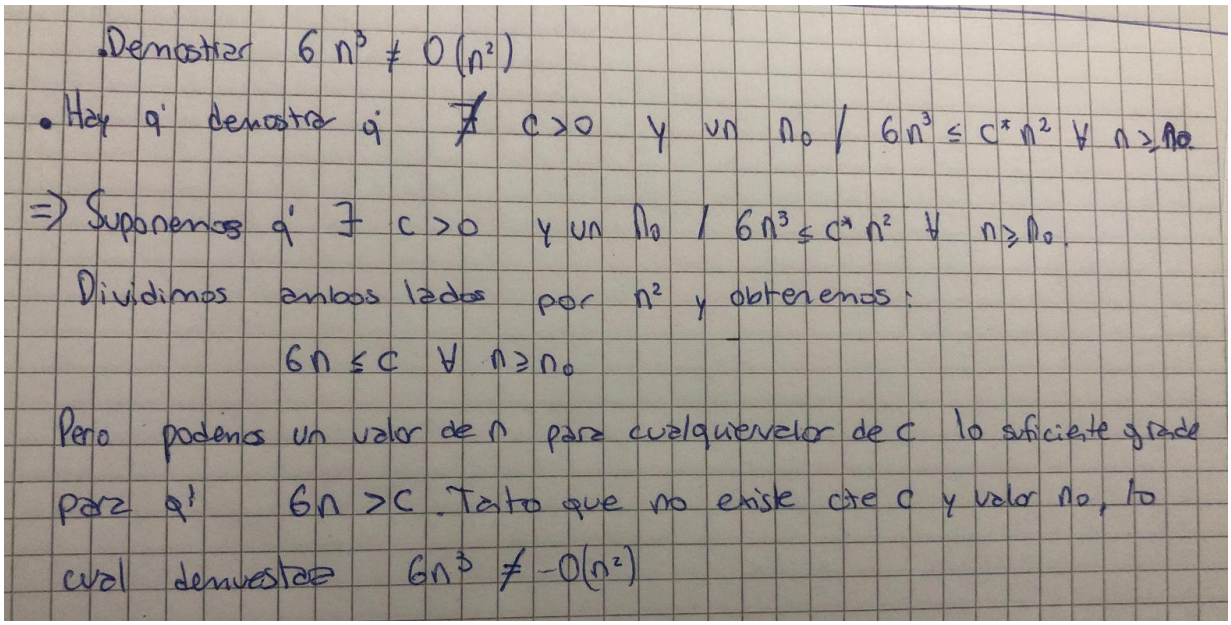
Aida Laricchia
13251

Ejercicio 1:

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



Ejercicio 2:

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

El mejor caso es cuando el pivote elegido divide a la lista en dos partes iguales,tamaño de la sublista se reduce a la mitad, en cada llamado.Esto nos garantiza una complejidad de tiempo de *O(n log n)*.

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(nlogn)

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

```
#Ejercicio 4

def Order_list(lista):

    # Encuentro el elemento de la mitad

    pivot = lista[len(lista) // 2]

    menores = []

    mayores = []

    # Recorro la lista y lleno las nuevas listas

    for i in range(len(lista)):

        if i == pivot:

            continue

        if lista[i] < lista[pivot]:

            menores.append(lista[i])

        else:

            mayores.append(lista[i])
```

```
    # Llamo a la recursividad con las nuevas listas

    menores_new = Order_list(menores)

    mayores_new = Order_list(mayores)

    # + concatena las listas y forma la nueva lista

    return menores_new + [lista[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.

```
def Contiene_Suma(A,n):

    for i in range(len(A)):

        if A[i]< n:

            for j in range(len(A)):

                if j!=i:

                    if A[i]+A[j]==n:

                        return True

    return False

#O(n^2) el tiempo de ejecucion aumenta  a medida que aumenta el tamaño del
array
```

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.

El ordenamiento bucket sort es un algoritmo de ordenamiento que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena individualmente y recursivamente este algoritmo para obtener casilleros con menos elementos.

1. Crear casilleros vacíos

2. Colocar cada elemento a ordenar en un único casillero

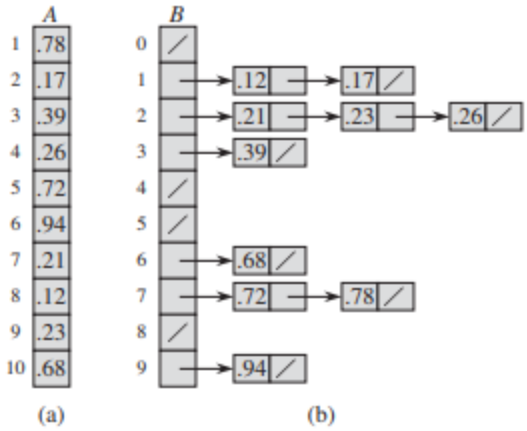
3. Ordenar individualmente cada casillero

4. devolver los elementos de cada casillero concatenados por orden

Una de las mayores desventajas de este algoritmo es que contiene una alta complejidad espacial.

Mejor Caso: O(n)

Caso Promedio y peor: O(n^2) todos los elementos en el mismo contenedor.

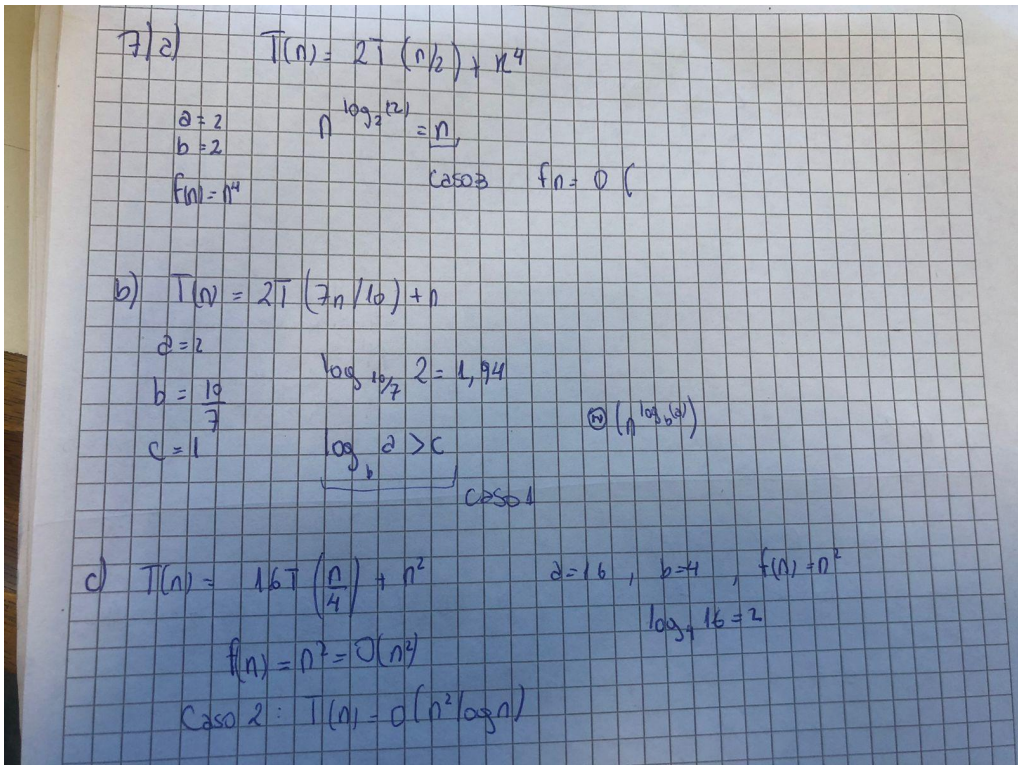


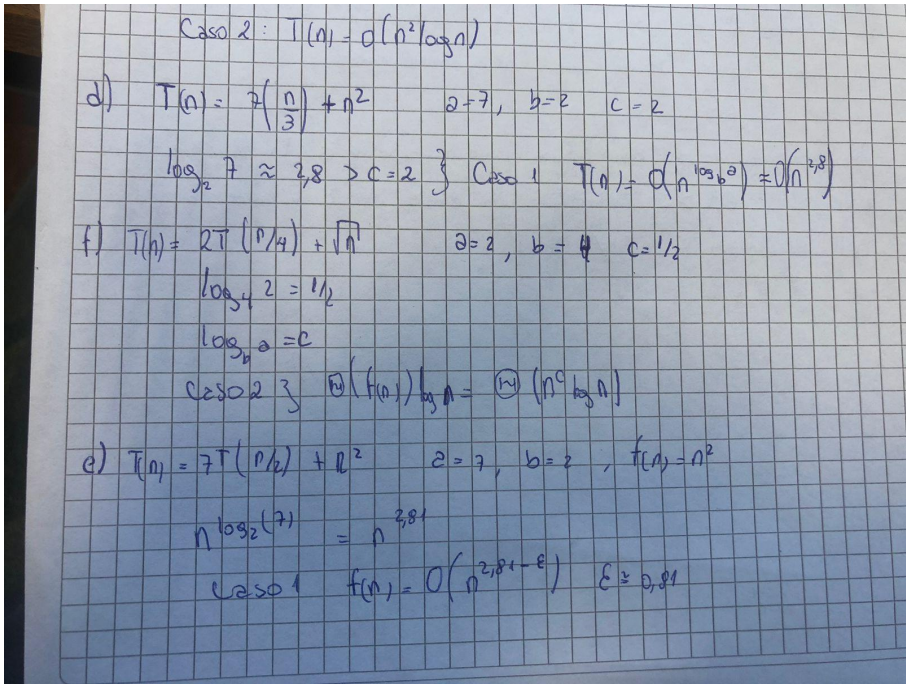

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊n A[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Ejercicio 7:

A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 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) + √n


7) a) T(n) = 2T(n/2) + n^4
a = 2
b = 2
f(n) = n^4
n^(log2(2)) = n
caso 3 f(n) = Ω(
b) T(n) = 2T(7n/10) + n
a = 2
b = 10/7
c = 1
log10/7 2 = 1,94
log_b a > c
caso 1
Θ(n^(log_b(a)))
c) T(n) = 16T(n/4) + n^2
a = 16, b = 4, f(n) = n^2
log4 16 = 2
f(n) = n^2 = O(n^2)
Caso 2: T(n) = Θ(n^2 log n)


Caso 2: T(n) = Θ(n^2 log n)
d) T(n) = 7(n/3) + n^2
a = 7, b = 2 c = 2
log2 7 ≈ 2,8 > c = 2 } Caso 1 T(n) = Θ(n^(log_b a)) = Θ(n^2,8)
f) T(n) = 2T(n/4) + √n
a = 2, b = 4 c = 1/2
log4 2 = 1/2
log_b a = c
caso 2 } Θ(f(n) log n) = Θ(n^c log n)
e) T(n) = 7T(n/2) + n^2
a = 7, b = 2, f(n) = n^2
n^(log2(7)) = n^2,81
caso 1 f(n) = O(n^(2,81 - ε)) ε = 0,01

A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca mas alla de algo1.py y linkedlist.py**~~
3. **Hacer una análisis por cada algoritmo implementado del caso mejor, el caso peor y una perspectiva del caso promedio.**