Ejercicio 1:

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

Para demostrar que 6n^3 ≠ O(n^2), se puede utilizar una demostración por contradicción. Supongamos que 6n^3 = O(n^2). Luego, se encuentra una contradicción mostrando que no existe una constante "c" y un tamaño mínimo de entrada "n0" tal que para todo "n" mayor que "n0", la complejidad temporal del algoritmo sea menor o igual a c * n^2. Por lo tanto, se concluye que 6n^3 no está acotado superiormente por O(n^2), lo que demuestra que 6n^3 ≠ 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 para el Quicksort (mínimo 10 elementos) es tener una lista desordenada y tomar el elemento del medio, esto último depende de la lista mas que nada otros tomar cualquier pivote sin hacer diferencia.

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): T(n) = n^2

En el caso del Quicksort lo que pasaría es que no importa que pivote se tome ya que el algoritmo dividira la lista en dos y pondra a todos los elementos menos el pivote en una lista izquierda o derecha (esto depende de como uno plantee el algoritmo con valores repetidos, ya sea <= o >=).

Insertion-Sort(A): T(n) = n

En el Insertion-Sort la lista solo se recorrera una vez debido a que nunca encontrara a un elemento que sea menor al de su izquierda.

Merge-Sort(A): T(n) = n(log(n))

Aquí no importa que elementos contenga la lista ya que el Merge-Sort dividira la lista a la mitad recursivamente y al final compara elementos iguales y los pondrá en la lista resultante en “orden” (esto depende también de como se implemente el algoritmo).

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

```python
#Ejercicio 4
def deleteByPosition(list, position, value):
    for i in range(len(list)):
        if i == position:
            list.remove(value)

def search(list, value, positionPivot):
    for i in range(positionPivot, len(list)):
        if list[i] == value:
            return i

longitud = int(input("Ingresar la longitud de la lista: "))
list = []
for i in range(longitud):
    list.append(randint(0,10))
print(list)

listMinLeft = []
listMinRight = []

if len(list)/2 == trunc(len(list)/2):
    #LISTA DE LONGITUD PAR
    positionPivote = trunc(len(list)/2) - 1
    pivote = list[positionPivote]
    for i in range(len(list)):
        if list[i] < pivote:
            if i < positionPivote:
                listMinLeft.append(list[i])
            else:
                listMinRight.append(list[i])
else:
    #LISTA DE LONGITUD IMPAR
    positionPivote = trunc(len(list)/2)
    pivote = list[positionPivote]
    for i in range(len(list)):
        if list[i] < pivote:
            if i < positionPivote:
                listMinLeft.append(list[i])
```

```
C: > Users > Juan Ignacio > OneDrive > Escritorio > Facultad > Algoritmos II 2023 > practicas > tp-complejidad
68              else:
69                  listMinRight.append(list[i])
70
71  print(listMinLeft)
72  print(listMinRight)
73  print("----------------------------")
74  contMaxLeft = 0
75  contMaxRight = positionPivote
76  while abs(len(listMinLeft) - len(listMinRight)) > 1:
77      if len(listMinLeft) > len(listMinRight):
78          if contMaxRight < len(list):
79              if list[contMaxRight] >= pivote:
80                  MaxRight = list[contMaxRight]
81                  del list[contMaxRight]
82                  position = search(list, listMinLeft[0], 0)
83                  deleteByPosition(list, position, listMinLeft[0])
84                  list.insert(0,MaxRight)
85                  list.append(listMinLeft[0])
86                  value = listMinLeft[0]
87                  listMinLeft.remove(value)
88                  listMinRight.append(value)
89          contMaxLeft += 1
90      else:
91          if contMaxLeft < positionPivote:
92              if list[contMaxLeft] >= pivote:
93                  MaxLeft = list[contMaxLeft]
94                  del list[contMaxLeft]
95                  position = search(list, listMinRight[0], positionPivote)
96                  deleteByPosition(list, position, listMinRight[0])
97                  list.append(MaxLeft)
98                  list.insert(0,listMinRight[0])
99                  value = listMinRight[0]
100                 listMinRight.remove(value)
101                 listMinLeft.append(value)
102         contMaxLeft += 1
103
104 print(listMinLeft)
105 print(listMinRight)
106 print(list)
```

Mejor caso: El mejor caso es de complejidad $\Omega(n)$ ya que solo recorreriamos la lista y verificaríamos que esta tiene la mitad de sus menores a la izquierda del pivote.

Peor caso y caso promedio: En estos casos la complejidad sería de n^2, ya que deberíamos recorrer la lista y además “balancearla”, es decir, colocar la mitad de los menores a la izquierda y tener en cuenta que el pivote siga siendo el mismo.

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.

```python
#Ejercicio 5
def Contiene_Suma(A,n):
    if A is None:
        return
    else:
        for i in range(len(A)-1):
            for j in range(i+1, len(A)):
                if A[i] + A[j] == n:
                    return True
    return

list = [2,6,8,3,1,0]
number = 7
output = Contiene_Suma(list,number)
if output is True:
    print("Existe el par")
else:
    print("No existe un par")
```

El costo computacional es de T(n) = n^2

Analizar casos de complejidad:

Peor Caso: O(n^2)

En el peor caso se tendrían que recorrer ambos bucles en su totalidad y no encontrar un par de números que sumados den n.

Caso Promedio: Θ(n^2)

Este sería el caso promedio ya que accederiamos a los dos bucles for para poder calcular la suma de los pares y que sea igual al entero pasado por parámetro. El primer for es para tomar un pivote de la lista y el segundo for es para poder sumar este “pivote” con los restantes elementos y vefificar si la suma es igual a n.

Mejor Caso: Ω(1)

El mejor caso del código anterior sería que la suma del par de enteros sean los elementos de las dos primeras posiciones ya que si esto sucedo solo se estaría accediendo una vez a ambos bucles.

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 algoritmo BucketSort es un método de ordenamiento que funciona dividiendo el conjunto de datos de entrada en varios “grupos” basados en su valor y luego ordenando cada cubeta individualmente utilizando otro algoritmo de ordenamiento. Después, los elementos de los grupos se unen en el orden correcto para producir el resultado final ordenado. Tiene una complejidad temporal de O(n+m), donde "n" es el número de elementos a ordenar y "m" es el número de grupos utilizados.

Peor caso: En el peor caso, la complejidad temporal de BucketSort es O(n^2), si todos los elementos caen en el mismo grupo y se utilizó un algoritmo de ordenamiento de complejidad O(n^2) para ordenar los elementos de cada grupo.

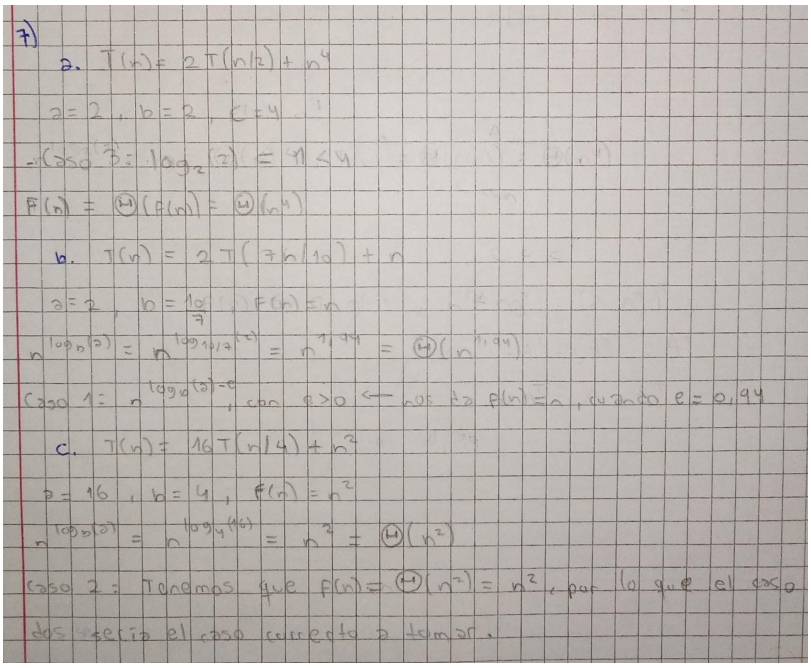
Mejor caso: En el mejor caso, la complejidad temporal de BucketSort es O(n), si cada elemento cae en un grupo diferente y se utiliza un algoritmo de ordenamiento de complejidad O(1) para ordenar los elementos de cada grupo. Sin embargo, este caso es poco común en la práctica.

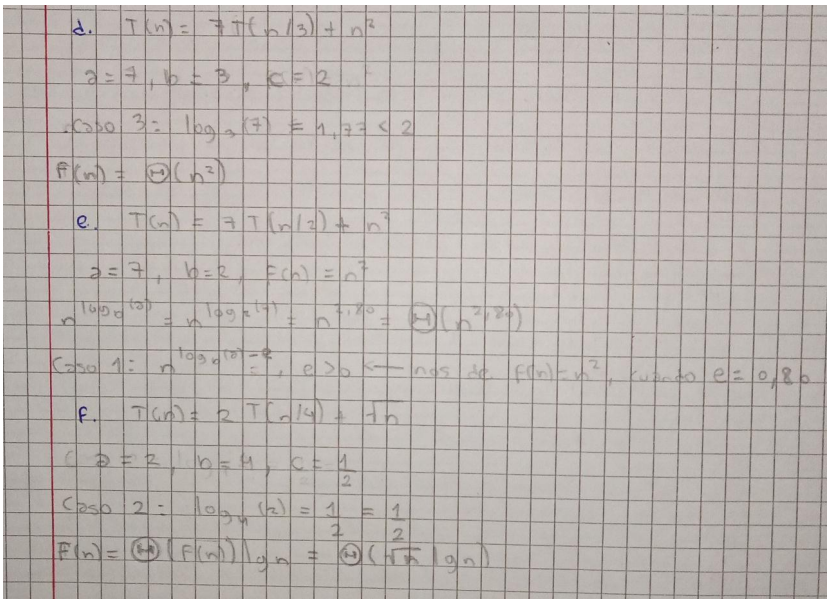
Caso Promedio: En el caso promedio, la complejidad temporal de BucketSort es O(n + k), donde "n" es el número de elementos a ordenar y "k" es el número de grupos utilizados.

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





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.**