Ejercicio 1:

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

Para que 6n^3 = O(n^2), deben existir constantes positivas c y n0 tales que:

6n^3 <= c*n^2 para n >= n0.

Despejando c obtenemos:

(6n^3) / (n^2) <= c

6n <= c

Para n = 1 esto es cierto, pero desde n = 2, en adelante, no lo es. Por lo que queda demostrado.

Ejercicio 2:

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

[1 2 4 5 3 7 8 10 9 6], donde el pivote siempre es el último elemento.

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: n^2

Insertionsort: n^2

Merge-sort: n logn

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

Mi función recibe una lista. Luego, se la ordena de menor a mayor, lo que tiene un orden de complejidad O(n logn). Vamos a elegir como elemento del medio a uno de los dos elementos más grandes de la lista. Si la lista tiene una cantidad par de elementos, el último elemento tiene una cantidad par de menores, por lo que, para ese caso, elegimos el penúltimo elemento de la lista, que por consecuencia tendrá una cantidad par de menores. Caso contrario, si la lista tiene una cantidad impar de menores, entonces elegimos el último elemento (el más grande), que tendrá una cantidad par de menores. Entonces nuestro algoritmo primero pregunta si la lista tiene una cantidad par o impar de elementos. Luego, guarda al elemento elegido en una variable y lo elimina de la lista. Finalmente, lo agrega en la posición del medio de la lista.

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.

Su orden de complejidad es de O(n logn). El costo computacional de ordenar la lista de menor a mayor es de n log n, pero una vez ordenada, encontrar la suma, en el mejor caso, será de orden constante si es que la suma nos la dieran el menor elemento y el mayor, y de orden n para el peor caso, que sería la suma de elementos consecutivos o que no existan dos elementos que sumen n. El costo sería T(n): n.logn + n en el peor caso.

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.

Vamos a ver el caso de ordenamiento de [2, 8, 5, 3, 9, 1] utilizando HeapSort.

Primero, debemos ordenar el arreglo para que nos quede con la estructura de Binary Heap.

Donde cada nodo es mayor a su hijo. El primer nodo es la raíz y se considera que es la posición 1. Si P es la posición de un nodo, 2P será la posición del hijo izquierdo y 2P + 1 será la posición del hijo derecho.

Ahora vamos con el ejemplo. Primero, hacemos que el arreglo tenga estructura Heap. Vemos que 9 es el hijo de 8, donde no se cumple que el papá 8 sea mayor al hijo 9. Por lo que los intercambiamos.

[2, 9, 5, 3, 8, 1]

Ahora el papá de 9 es 2, por lo que los intercambiamos.

[9, 2, 5, 3, 8, 1]

2 es menor a su hijo 8, entonces los intercambiamos.

[9, 8, 5, 3, 2, 1]

Ahora que ya tenemos un Binary Heap, empezamos.

Intercambiamos el primer elemento por el último.

[1, 8, 5, 3, 2, 9]

Donde 9 ya tiene su posición final y “reducimos” la cantidad del arreglo en 1.

Para mantener la propiedad de Binary Heap, intercambiamos a 1 por su hijo 8.

[8, 1, 5, 3, 2, 9]

Ahora a 1 por 3.

[8, 3, 5, 1, 2, 9]

Ya tenemos un Heap. Ahora intercambiamos el primer elemento por el último.

[2, 3, 5, 1, 8, 9]

Donde 8 tiene ya su posición final y volvemos a reducir en 1 el arreglo.

Intercambiamos a 2 por su hijo 5 para mantener la propiedad.

[5, 3, 2, 1, 8, 9]

Ahora intercambiamos el primero por el último.

[1, 3, 2, 5, 8, 9]

Donde 5 ya tiene su posición final.

Intercambiamos 1 por 3 para mantener la propiedad.

[3, 1, 2, 5, 8, 9]

Intercambiamos el primero por el último.

[2, 1, 3, 5, 8, 9]

3 ya tiene su posición final.

Se cumple ya con la propiedad, por lo que intercambiamos el primero por el último.

[1, 2, 3, 5, 8, 9]

2 ya tiene su posición definitiva. Como solo queda un elemento, el arreglo ya está ordenado y finaliza HeapSort.

[1, 2, 3, 5, 8, 9]

En cualquiera de los casos, HeapSort tiene una complejidad de n logn. A lo sumo, habrán casos en los que se tenga que hacer más operaciones elementales, pero que no cambian la complejidad del algoritmo.

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: $T(n) = a\ T(n/b) + n^c$

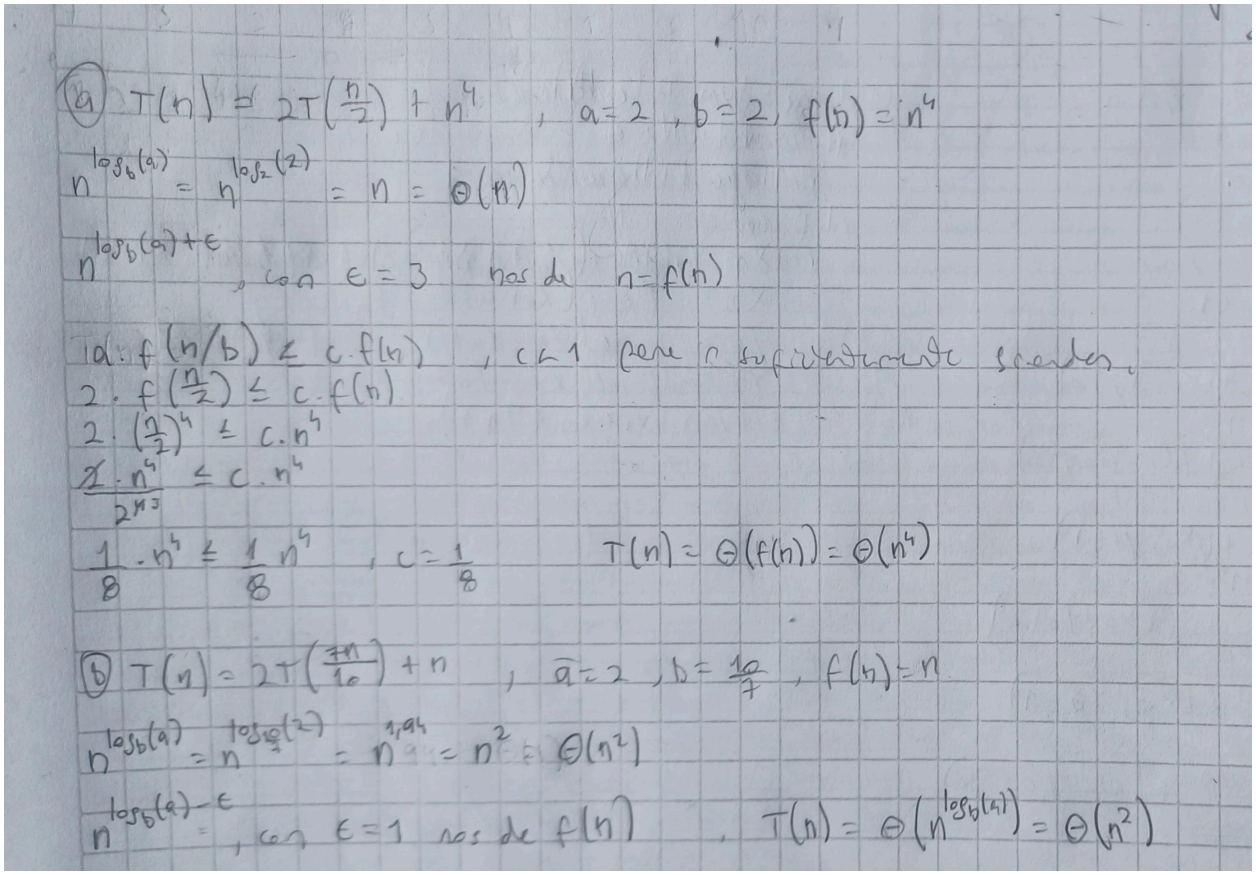
a. $T(n) = 2T(n/2) + n^4$
b. $T(n) = 2T(7n/10) + n$
c. $T(n) = 16T(n/4) + n^2$

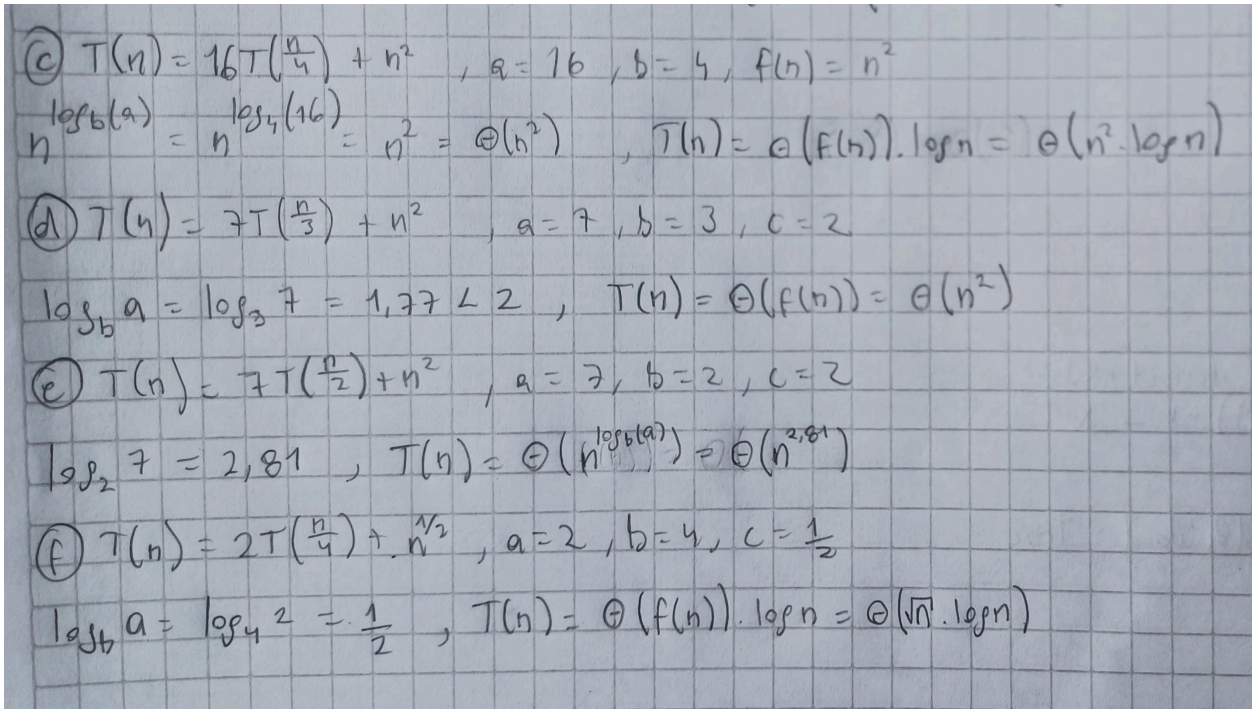
d. $T(n) = 7T(n/3) + n^2$
e. $T(n) = 7T(n/2) + n^2$
f. $T(n) = 2T(n/4) + \sqrt{n}$

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

2. $T(n) = 2T(7n/10) + n$. $\Theta(n^{1.94})$
3. $T(n) = 7T(n/3) + n^2$. $\Theta(n^2)$
4. $T(n) = 16T(n/4) + n^2$. $\Theta(n^2 \log n)$
5. $T(n) = 7T(n/2) + n^2$. $\Theta(n^{2.81})$

6. $T(n) = 2T(n/2) + n^4$. $\Theta (n^4)$





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.**