Ejercicio 1:

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

6n^3 != O(n^2)

La notación big-O solo tiene en cuenta las variables n y su orden (potencia). Entonces 6n^3 en notación big-O sería O(n^3). Si volvemos a la igualdad anterior:

O(n^3) != O(n^2) lo cual es verdadero

Ejercicio 2:

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

El orden de complejidad del mejor caso de quickSort es igual al caso promedio, es decir, 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?

El tiempo de ejecución para Quicksort(A) si los elementos de A son todos iguales será de O(n^2) porque el método de Quicksort elige un pivote y manda a la derecha los mayores y a la izquierda los menores, entonces, si todos los elementos son iguales los trataría de la misma manera (por ejemplo los manda a todos a la derecha) y esto llevaría al Quicksort al peor de los casos n^2

El tiempo de ejecución para Insertion-Sort(A) si los elementos de A son todos iguales será de O(n) ya que a los elementos iguales los deja en su posición original

El tiempo de ejecución de Marge-Sort(A) si los elementos de A son todos iguales será de O(n log(n)) porque el método divide a la lista en sublistas y a lo hace igual sean como sean los elementos

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 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.

La complejidad del algoritmo implementado es de O(1) en el mejor caso, que sería el caso donde los primeros dos elementos sumados dan n.

En el peor caso seria de O(n^2) que sería el caso donde los dos elementos que sumados dan n esten en las últimas posiciones

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.

Bucket Sort:

Supongamos que tenemos una lista de números que queremos ordenar. En el algoritmo de Bucket Sort, dividimos los elementos de la lista en un número finito de "cubetas" (buckets) según algún criterio (por ejemplo, el valor de los elementos). Luego, cada cubeta se ordena individualmente, y finalmente se combinan todas las cubetas para obtener la lista ordenada.

Considera la siguiente lista de números: `[0.42, 0.64, 0.15, 0.27, 0.91, 0.45, 0.67, 0.08]`. Si queremos ordenar estos números utilizando Bucket Sort, podríamos dividirlos en cubetas según la parte decimal y luego ordenar cada cubeta.

- El tiempo de ejecución de Bucket Sort depende de la cantidad de elementos en la lista y el número de cubetas utilizadas. En el mejor caso, si los elementos están uniformemente distribuidos entre las cubetas, su complejidad es lineal, O(n + k), donde n es el número de elementos y k es el número de cubetas. En el peor caso, si todas las claves se agrupan en una sola cubeta, la complejidad es cuadrática, O(n^2).

Heap Sort:

Heap Sort es un algoritmo de ordenamiento basado en la estructura de datos del montículo (heap). Comienza construyendo un montículo máximo (para ordenamiento ascendente) o un montículo mínimo (para ordenamiento descendente) a partir de los elementos de la lista. Luego, extrae repetidamente el elemento máximo (o mínimo) del montículo y lo coloca al final de la lista, reduciendo el tamaño del montículo en cada paso. El proceso se repite hasta que el montículo esté vacío.

Supongamos que tenemos la siguiente lista de números: `[12, 11, 13, 5, 6, 7]`. Para ordenar estos números utilizando Heap Sort, primero construimos un montículo máximo. Luego, extraemos repetidamente el elemento máximo y lo colocamos al final de la lista hasta que el montículo esté vacío.

- La complejidad de tiempo de Heap Sort es O(n log n), donde n es el número de elementos en la lista. Tanto la construcción del montículo como las operaciones de extracción requieren logarítmicamente el tiempo proporcional al número de elementos en el montículo.

Radix Sort:

Radix Sort es un algoritmo de ordenamiento no comparativo que ordena los elementos procesando los dígitos individuales de los números. Se ordenan los elementos dividiéndolos en "radix" (base) y ordenándolos en cada paso según los dígitos correspondientes. Este proceso se repite hasta que todos los dígitos han sido considerados.

Supongamos que tenemos la siguiente lista de números: `[170, 45, 75, 90, 802, 24, 2, 66]`. Para ordenar estos números utilizando Radix Sort, podríamos comenzar ordenando los números según sus unidades, luego según sus decenas, y así sucesivamente, hasta que todos los dígitos hayan sido considerados.

- La complejidad de tiempo de Radix Sort es O(n * k), donde n es el número de elementos en la lista y k es el número de dígitos del número más largo. La complejidad de espacio es O(n + k) para almacenar los elementos y las cubetas.

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