Ejercicio 1:

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

Resolución: Podemos utilizar la definición de O grande: 6n^3 ≠ O(n^2) si y solo si no existe una constante positiva c y un valor n_0 tal que para todo n ≥ n_0 se cumpla que $6n^3 \leq c * n^2$.

Entonces, procedemos a verificar la igualdad:

$\mathbf{6n^3 = c*n^2}$

Dividimos ambos miembros por n^2

6n = c

Y esto resulta falso, ya que no hay ninguna constante c tal que pueda cumplir la igualdad para todo n. En conclusión, $6n^3 \neq \boldsymbol{O}(n^2)$.

Ejercicio 2:

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

Para el mejor caso, el array debería estar ya ordenado, así no intercambia elementos. Por ejemplo:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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 y MergeSort tendrían un tiempo de ejecución de O(nlog(n)), e Insertion-Sort de O(n^2)

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

```python
def halfSort(L):
    size = length(L)
    currentNode = L.head
    midNode = L.head
    k = 1
    smallerNodes = 0
    midPosition = 0
    position = 0
    smallerNodeCount = 0

    while (k <= size/2):
        midNode = midNode.nextNode
        midPosition = k
        k += 1


    while (currentNode != None):
        if (currentNode.value < midNode.value):
            smallerNodes += 1
            if(position < midPosition):
                smallerNodeCount += 1
        currentNode = currentNode.nextNode
        position += 1

    currentMidNode = midNode.nextNode
    currentNode = L.head
    if(smallerNodeCount == math.trunc(smallerNodes/2)):
        return L
    else:
        while(currentMidNode != None and smallerNodeCount < math.trunc(smallerNodes/2)):
            if(currentMidNode.value < midNode.value):
                if(currentNode.value > currentMidNode.value):
                    aux = currentNode.value
                    currentNode.value = currentMidNode.value
                    currentMidNode.value = aux
                    currentNode = currentNode.nextNode
                    smallerNodeCount += 1
            currentMidNode = currentMidNode.nextNode
```

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.

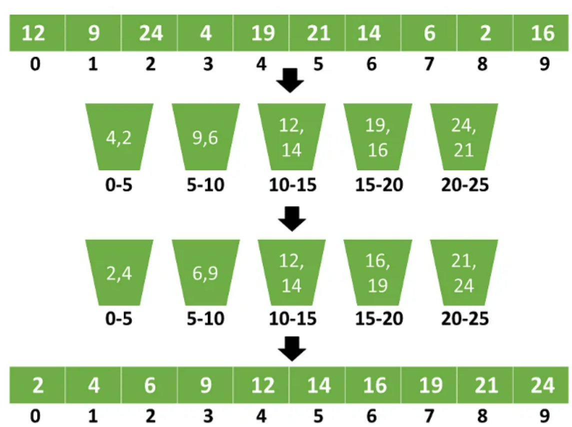
```
def contieneSuma(L, n):
    currentNode = L.head
    while(currentNode != None):
        currentNodeAux = currentNode
        while(currentNodeAux != None):
            if(currentNode.value + currentNodeAux.value == n):
                return True
            currentNodeAux = currentNodeAux.nextNode
        currentNode = currentNode.nextNode
    return False
```

Su complejidad es de O(n^2)

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.

BucketSort es un algoritmo de ordenamiento que utiliza un enfoque basado en contenedores para ordenar un conjunto de elementos. En lugar de comparar los elementos entre sí, BucketSort divide los elementos en contenedores, también llamados "buckets", y luego ordena cada bucket individualmente. Una vez que se han ordenado los buckets, los elementos se combinan en orden para producir el conjunto de elementos ordenado completo.



El orden de BucketSort depende de la forma en que se implemente y de cómo se distribuyan los elementos en los buckets. En el caso promedio, si los elementos se distribuyen uniformemente en los buckets, el orden de BucketSort es O(n+k), donde n es el número de elementos en el conjunto y k es el número de buckets. En el mejor caso, si todos los elementos caen en un solo bucket, el orden sería O(n log n) debido al algoritmo de ordenamiento utilizado dentro de cada bucket. En el peor caso, si todos los elementos caen en el mismo bucket, el orden sería O(n^2) si se utiliza un algoritmo de ordenamiento cuadrático dentro de cada bucket. En general, BucketSort es útil cuando se trabaja con datos distribuidos uniformemente, como en el ejemplo anterior, pero puede ser menos eficiente si los datos no están uniformemente distribuidos.

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

Si tenemos un algoritmo cuya ecuación de recurrencia es:

$$T(n) = A * T(n / B) + \mathcal{O}(n\text{^}C)$$

A: cantidad de llamados recursivos

B: proporción del tamaño original con el que llamamos recursivamente

$\mathcal{O}$(n^C): el costo de *partir y juntar* (todo lo que no son llamados recursivos)

a. T(n) = 2T(n/2) + n^4

   a = 2, b = 2 y f(n) = n^4. Entonces, n^logb(a) = n^log2(2) = n. Como f(n) es del mismo orden que n^logb(a), la complejidad es Θ(n^logb(a) log n) = Θ(n log n).

b. T(n) = 2T(7n/10) + n

   a = 2, b = 10/7 y f(n) = n. Entonces, n^logb(a) = n^log7/10(2) ≈ n^0.643. Como f(n) es mayor que n^logb(a), la complejidad es Θ(f(n)) = Θ(n).

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

a = 16, b = 4 y c = 2. Entonces, n^c = n^2. Como a es del mismo orden que n^c/b^c, la complejidad es Θ(n^c log n) = Θ(n^2 log n).

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

a = 7, b = 3 y c = 2. Entonces, n^c = n^2. Como a es mayor que n^c/b^c, la complejidad es Θ(a^n) = Θ(7^n)

e. $T(n) = 7T(n/2) + n^2$

a = 7, b = 2 y f(n) = n^2. Entonces, n^logb(a) = n^log2(7) ≈ n^2.81. Como f(n) es menor que n^logb(a), la complejidad es Θ(n^logb(a)) = Θ(n^2.81).

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

a = 2, b = 4 y c = 1/2. Entonces, n^c = √n. Como a es mayor que n^c/b^c, la complejidad es Θ(a^n) = Θ(2^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.**