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) ?

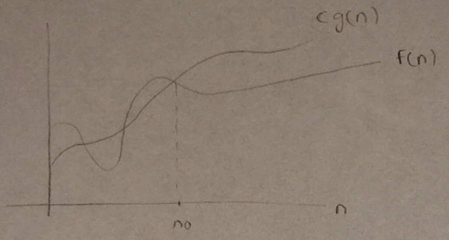
Ejercicio 3:

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?

TP1 - Complejidad

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

Según la definición de la notación Big O, se dice que $T(n)$ es $O(f(n))$ si existen constantes positivas c y n_0 tal que: $T(n) \leq cf(n)$ cuando $n \geq n_0$



$f(n) = O(g(n))$

$6n^3 = O(n^2)$ si existe una constante positiva C y un número entero positivo n_0 tal que $|6n^3| \leq C \cdot |n^2|$ para todo $n > n_0$

$$\frac{|6n^3|}{n^2} \leq C \frac{|n^2|}{n^2} \rightarrow 6n \leq C \quad \forall n > n_0$$

Como $6n$ crece indefinidamente a medida que n aumenta y es una constante finita, no es posible encontrar una constante C y un n_0 tal que la desigualdad $6n \leq C$ → sea cierta. Por lo cual $6n^3 \neq O(n^2)$

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

Si usamos un array ordenado la partición siempre dividirá el array en 2 subarrays de tamaño casi igual en cada iteración

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ↳ Esto significa que la profundidad del árbol de llamados recursivas será mínima y el algoritmo de Quicksort tendrá complejidad $O(n \log n)$

3 ¿Cuál es el tiempo de ejecución de la estrategia Quicksort(A), Insertion-Sort(A) y Mergesort(A) cuando todos los elementos de A tienen el mismo valor?

- Quicksort: $O(n^2)$ debido a que todos los elementos son iguales, cada partición seguirá siendo desequilibrada
- InsertionSort: $O(n)$
- Mergesort: $O(n \log n)$

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
from linkedlist import*
from ordAvanzado import*


def contieneSuma(A,n):
    terminar = False
    encontro = False
    num = A.head
    while terminar == False and num.nextNode != None:
        valorBusc = n - num.value
        num2 = num.nextNode
        while encontro == False:
            if num2.value == valorBusc:
                encontro = True
                terminar = True
                return True
            else:
                num2 = num2.nextNode
                if num2 == None:
                    break
        num = num.nextNode
```

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 divide el arreglo de entrada en un número finito de cubetas. Cada cubeta se ordena individualmente, ya sea utilizando otro algoritmo de ordenamiento o aplicando recursivamente el algoritmo de BucketSort. Una vez que todas las cubetas están ordenadas, los elementos se concatenan para formar el arreglo ordenado.

<u>Descripción</u>:

1. Particionamiento: divide el arreglo de entrada en un numero fijo de cubetas
2. Distribución: distribuye los elementos del arreglo de entrada en las cubetas correspondientes

3. Ordenamiento: ordena cada cubeta individualmente, ya sea utilizando otro algoritmo de ordenamiento o aplicando BucketSort recursivamente
4. Concatenación: concatena las cubetas ordenadas para formar el arreglo ordenado

Ejemplo caso promedio:

Supongamos que tenemos un arreglo de números de punto flotante que van de 0 a 1

1. Particionamiento: dividir el rango [0,1] en n cubetas de igual tamaño, donde n es el tamaño del arreglo de entrada
2. Distribución: colocar cada elemento del arreglo de entrada en la cubeta correspondiente según su valor
3. Ordenamiento: ordenar cada cubeta individualmente. Dado que los números son de punto flotante y estan distribuidos uniformemente, podemos utilizar un algoritmo de ordenamiento eficiente como el ordenamiento por inserción para cada cubeta
4. Concatenación: concatenar las cubetas ordenadas para formar el arreglo ordenada

Orden en caso promedio: O(n+k) donde n es el número de elementos y k el número de cubetas

Orden en mejor caso: O(n+k) donde n es el número de elementos y k el número de cubetas

Orden peor caso: O(n^2), si todos los elementos se colocan en una sola cubeta y el algoritmo de ordenamiento utilizado para cada cubeta tiene una complejidad temporal peor caso de O(n^2)

El BucketSort es eficiente para ordenar un conjunto grande de elementos con una distribución uniforme, pero puede no funcionar bien si la distribución de entrada está sesgada.

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) $T(n) = 2T(n/2) + n^4$

$n^{\log_b a} = n^{\log_2 2} = n$

caso 2 → $f(n) \neq n$

caso 1 → $n^{1-\varepsilon} \overset{?}{=} n^4$ x

caso 3 → $n^{1+\varepsilon} \overset{?}{=} n^4$ cuando $\varepsilon = 3$

$n^4 = n^4$

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

$a = 2$, $b = 10/7$ $F(n) = n$, $C = 1$

$n^{\log_{\frac{10}{7}} 2} = n^{1,94}$ Aplicamos caso 1 pq

$\log_b a > c \rightarrow 1,94 > 1$

Caso 1: $O(n^{1,94})$

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

$n^{\log_4 16} = n^2$ Por lo tanto para el caso 2

$T(n) = \Theta(n^2 \lg n)$

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

$n^{\log_3 7} = n^{1,77}$ $f(n) = n^2$ $1,77 < 2$ caso 3

$T(n) = \Theta(n^2)$

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

$a = 7$ $n^{\log_b a} = n^{\log_2 7} = n^{2,81} > n^2$ caso 1

$b = 2$ para el caso 1 $T(n) = O(n^{2,81})$

$F(n) = n^2$

f) $T(n) = 2T(n/4) + n^{1/2}$

$a = 2$ $n^{\log_4 2} = x$ $n^{1/2} = F(n) \rightarrow$ caso 2 ya que

$b = 4$ $4^x = 2$ $T(n) = \Theta(n^{1/2} \lg n)$

$F(n) = n^{1/2}$ $(2^2)^x = 2$

$2x = 1$

$x = 1/2$