Ejercicio 1:

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

Respuesta:

Siendo $6n^3$ escrito tambien de la forma $6n^3 + 0n^2 + 0n + 0$, podemos decir que el peor de los casos de este algoritmo (O(n)) nunca va a ser menor a la complejidad cúbica del problema principal

Eercicio 2:

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

Respuesta:

El mejor de los casos para un algoritmo de ordenamiento Quicksort quiere decir que no tiene que reposicionar ningún elemento del array inicial, por lo que, en el mejor de los casos cada elemento del array está ordenado con respecto al pivote (cualquiera sea) y basándonos en eso, quiere decir que es un array previamente ordenado, por ejemplo:

[0,1,2,3,4,5,6,7,8,9]

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?

Respuesta:

Al array **A** estar lleno del mismo elemento, quiere decir que está ordenado como punto inicial, por lo que en todas las estrategias, es siempre el mejor de los casos, por lo que:
En QuickSort, el tiempo de ejecución va a ser: **O(A log A)**

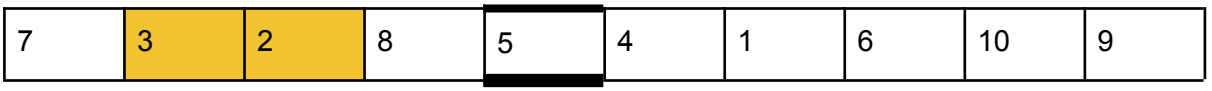
En InsertionSort, el tiempo de ejecucion va a ser: **O(A)**

En MergeSort, el tiempo de ejecucion va a ser: **O(A log A)**

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



Respuesta:

```
def OrdenPorIzquierda(L, desde):
    longituds2 = length(L) // 2
    mitad = access(L, length(L) // 2)
    cnode = L.head
    c = 0
    for i in range(longituds2):
        if cnode.value < mitad:
            c += 1
        cnode = cnode.nextNode
    if c < longituds2 // 2:
        if buscarmayor(L, mitad, desde) != None:
            mayor = buscarmayor(L, mitad, desde)
            ubicacionmayor = search(L, mayor)
            L = swichValores(L, longituds2, ubicacionmayor)
            OrdenPorIzquierda(L, desde + 1)
    if c > longituds2 // 2:
        if buscarmenor(L, mitad, desde) != None:
            menoraux = buscarmenor(L, mitad, desde)
            ubicacionmenor = search(L, menoraux)
            L = swichValores(L, longituds2, ubicacionmenor)
            OrdenPorIzquierda(L, desde + 1)


def buscarmayor(L, n, desde):
    for i in range(desde, length(L) // 2):
        if access(L, i) > n:
            return access(L, i)
    return None


def buscarmenor(L, n, desde):
    for i in range(desde, length(L) // 2):
        if access(L, i) < n:
            return access(L, i)
    return None
```

```
def swichValores(L, n, m):
    pr = access(L, n)
    seg = access(L, m)
    update(L, pr, m)
    update(L, seg, n)
    return L
```

La estrategia utilizada es:
Selecciono el valor del elemento que está en la mitad de la lista y calculo cuantos elementos menores a este valor debe tener a su izquierda en la lista. Una vez seleccionado éste elemento, cuento cuantos elementos menores tiene a su izquierda, si el contador de elementos me da menor a la mitad del tamaño de la lista, entonces encuentro un valor mayor a éste (sea o no el mayor de la mitad de la lista), y “swapeo” lugares, por lo que, una vez intercambiadas las posiciones de los valores, tomo el nuevo valor que ahora está en el medio de la lista y hago recursivamente el mismo proceso, hasta que en la mitad de la lista, quede un elemento que cumpla la condición del ejercicio

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 Contiene_Suma(A, n):
    cnode = A.head
    for i in range(length(A)):
        cnode2 = cnode.nextNode
        for j in range(i + 1, length(A)):
            if (cnode.value + cnode2.value) == n:
                return True
            cnode2 = cnode2.nextNode
        cnode = cnode.nextNode
    return False
```

El orden de complejidad temporal es $O(n^2)$ en el peor de los casos y en el mejor de los casos (los 2 primeros números de la lista suman n) es O(1)

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:
BucketSort es un sistema de Ordenamiento que analiza cada elemento de una lista y las asigna a una coleccion, dependiendo de ciertas condiciones, este sistema de ordenamiento siempre tiene una complejidad de O(n) En todos sus casos (promedio, mejor o peor), la complejidad que aumenta es la espacial, ya que crea en total 3 elementos de gran tamaño.

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) + √n

A) 2T(n/2) + n^4

a= 2, b=2 c=4 f(n)=n^4

$O(n^{log2\ (2)})$

$O(n)<n^4$

Entonces, tercer caso(ya que 2T(n/2)<n^4

$\Omega(n^{log2\ (2)+e}) = n^4$

$\Omega(n^{1+e}) = n^4$

$\Omega(n^4) = n^4$ con $e = 3$

Al ser tercer caso, analizamos regularidad

$a(fn/b) = 2f(n/2)$

$= 2f(n/2)^4$

$= 1/8n^4 <= 1/8n^4$

$c=⅛<1$, se cumple la regularidad

Complejidad= Θ(n^4)

B) 2T(7n/10) + n

a= 2, b=7/10 c=1 f(n)=n

$O(n^{log10/7\ (2)}) => (n^{1,94})>f(n)$

Primer Caso

$O(n^{log7/10\ (2)-e}) = fn\ e>0$

$e = 0,94 => O(n^{1,94-0,94}) = n$

$O(n)=n$

Complejidad=$O(n^2)$

C) 16T(n/4) + n²

a= 16, b=4 c=2 f(n)=n²

$O(n^{log4(16)}) = f(n)$

Segundo caso

$O(n^{log4\ (16)} lgn) = O(n^2\ lgn)$

Complejidad = $O(n^2\ lgn)$

D) 7T(n/3) + n²

a= 7, b=3 c=2

log(3)7

1,77<2

Caso 3

Complejidad = $O(n^2)$

E) 7T(n/2) + n²

a= 7, b=3 c=2

log(2)7

2,8>2

caso 1

Complejidad = O(n³)

F) 2T(n/4) + sqrt(n)

a= 7, b=3 c=½

log(4)2

½=½

caso 2

Complejidad = O($n^{1/2}$)