Trabajo Práctico Complejidad

Ejercicio 1:

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

Respuesta: Para demostrar este enunciado utilizamos la definición de la notación Asintótica Superior que dice:

$T(n)$ es $O(f(n))$ si existen constantes positivas c y n_0 tal que:

$$T(n) \leq cf(n) \text{ cuando } n \geq n_0$$

Si lo aplicamos al enunciado quedaría:

6n^3 <= cn^2

Pero si nos fijamos bien no se puede encontrar una constante c >= 0 que cumpla con la desigualdad, es decir el 6n^3 siempre se va a hacer más grande por lo tanto no se cumple con la desigualdad.

Ejercicio 2:

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

Respuesta: Para que se presente el mejor caso en el Quicksort el array no debe contener elementos repetidos. Además podemos sumarle la condición del pivote y es que a la hora de elegirlo sea el elemento que se encuentre en el medio (o que no sea el primer o último elemento), esto garantiza en los casos que la lista se encuentre ordenada de manera ascendente o descendente no nos queden sublistas desequilibradas.

La complejidad en el mejor caso es de **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?

Respuesta:

-QuickSort(A): El tiempo de ejecución en el quicksort si todos los elementos son iguales es de **O(n^2)**

-InsertionSort(A): En el insertionsort si todos los elementos tienen el mismo valor no se intercambian de lugar, es decir se dejan en la misma posición por lo tanto la complejidad va a ser de **O(n).**

-MergeSort(A): La complejidad en el mergesort al tener elementos iguales va a ser de **O(n log n)** ya que las sublistas se hacen de igual manera aunque se encuentren elementos repetidos.

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

El algoritmo está explicado en el archivo.py

```
#Ejercicio 4
def middlesort(A):
    #Calculamos primero la longitud del Array,la posición del medio y el elemento posición
    length = len(A)
    middlepos = math.trunc(length/2)
    middle = A[middlepos]

    #Luego iniciamos varios contadores
    #minorscounter va a ser igual a la cantidad de menores en todo el array,minorsmiddlecount
    #minorspos va a ser una linkedlist con las posiciones de los menores y minorsposinverted
    minorspos = LinkedList()
    minorsposinverted = LinkedList()
    minorscounter = 0
    minorsmiddlecounter = 0
    j = 0

    #En este bucle verificamos los elementos menores y sumamos e insertamos respectivamente
    for i in range(0,length):
        if A[i] < middle:
            if i < middlepos:
                minorsmiddlecounter +=1
            minorscounter += 1
            insert(minorspos,i,j)
            add(minorsposinverted,i)
            j +=1

    #flag es la mitad de los menores
    flag = math.trunc(minorscounter/2)
```

```python
    # Caso 1 : si la mitad de los menores es igual a la cantidad de menores que hay a la izquierda retornamos la lista.
    if flag == minorsmiddlecounter:
        return A

    # Caso 2: Si la mitad de los menores es menor que la cantidad de menores a la izq,hacemos un cambio de elementos en
    #y menores utilizando como referencia la posicion de los menores con la linkedlist minorspos
    if flag < minorsmiddlecounter:
        currentnode = minorspos.head
        for i in range(middlepos+1,length):
            if A[i] > middle:
                aux = A[currentnode.value]
                A[currentnode.value] = A[i]
                A[i] = aux
                minorsmiddlecounter -=1
                if flag == minorsmiddlecounter:
                    return A
                else:
                    currentnode = currentnode.nextNode

    #Caso 3: Si la mitad de los menores es mayor que la cantidad de menores a la izq,hacemos un cambio de elementos com
    #Pero con la linkedlist minorsposinverted, ya que queremos las posiciones de los menores a la derecha del elemento
    if flag > minorsmiddlecounter:
        currentnode = minorsposinverted.head
        for i in range(0,middlepos-1):
            if A[i] > middle:
                aux = A[currentnode.value]
                A[currentnode.value] = A[i]
                A[i] = aux
                minorsmiddlecounter += 1
                if flag == minorsmiddlecounter:
                    return A
                else:
                    currentnode = currentnode.nextNode


    return
```

Complejidad: En el mejor, peor o caso promedio la complejidad del algoritmo va a ser de O(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
def contiene_suma(A,n):
    for i in range(0,len(A)-1):
        s = A[i]
        j = 1
        for j in range(j,len(A)-1):
            if s + A[j] == n:
                return True
        j += 1
    return False
```

El coste computacional es de **O(n^2)**

Mejor caso: El mejor caso es que los dos primeros elementos sean igual al n que nos dan como parámetro

Caso Promedio: El caso promedio es que los dos elementos que al sumarlos nos den el n se encuentren en la mitad de la lista.

Peor caso: El peor caso sería que los dos elementos se encuentren al final de la lista o que no se encuentren.

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.

Radix Sort

El radix sort es un algoritmo de ordenamiento no comparativo, su particularidad es que los ordena basándose en los dígitos individuales.

El procedimiento es el siguiente:

1- Primero busca el número más grande de la lista (esto se realiza ya que al ordenar la lista a partir de los dígitos individuales, al tener el número más grande sabemos cuánto es el rango o longitud más grande)

2-Separa los elementos de la lista en dígitos individuales.

3- A diferencia del bucketsort, empezamos con los dígitos menos significativos (unidades) y los ordena según ese dígito. Luego continúa hasta llegar con los dígitos más significativos (decenas,centenas etc.)

4- Se combinan las listas.

Ejemplo gráfico:

[170,45,75,90,24,2] --> Array

0: 170,90
1:
2: 2
3:
4: 24
5: 45,75
6:
7:
8:
9:

Ordenamos y el array queda asi: [170,90,2,24,45,75]

La primera iteración tomamos como dígito el último, es decir la unidad.

Ahora la segunda iteración tomamos las decenas.

0: 02
1:
2: 24
3:
4: 45
5:
6:
7: 170,75
8:
9: 90

Ordenamos el array y queda así: [2,24,45,170,75,90]

Y por último el dígito más significativo que es la centena.

0: 2,24,45,75,90
1: 170
2:
3:
4:
5:
6:
7:
8:
9:

Ordenamos el array y queda así: [2,24,45,75,90,170]

Por ende la lista quedó ordenada.

El **orden de complejidad del Radix Sort** es de **O(kn)** siendo n la longitud de la lista y k es la longitud máxima de los elementos (es decir la mayor cantidad de dígitos en un número, en el ejercicio anterior sería de 3 el k).

Mejor Caso: El mejor caso en el radix sort es si todos los elementos tienen la misma longitud por lo tanto la complejidad del algoritmo será de **O(n)**

Peor Caso: El peor caso del radix sort ocurre cuando los dígitos más significativos de la lista son el mismo y el menos significativo son distintos. La complejidad sería de **O(kn)**

Caso Promedio: El caso promedio del radix sort es si tenemos que ordenar una lista como la anterior. Por lo tanto su complejidad va a ser **O(kn).**

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

Ecuaciones de Recurrencia con Método Maestro Simplificado:

A,D y F

Ejercicio 7.) Ecuaciones de recurrencia con Método Maestro Simplificado

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

$a = 2$, $b = 2$, $c = 4$

$\log_b a = \log_2 2 = \boxed{1 < 4}$

Como $c > \log_b a$, esta ecuación se puede resolver con el primer caso. Por lo tanto:

$T(n) = O(f(n)) = \boxed{O(n^4)}$

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

$a = 7$, $b = 3$, $c = 2$

$\log_b a = \log_3 7 = \boxed{1{,}77 < 2}$

Como $c > \log_b a$, se cumple el 3er caso

$T(n) = O(f(n)) = \boxed{O(n^2)}$

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

$a = 2$, $b = 4$, $c = \frac{1}{2}$

$\log_b a = \log_4 2 = \boxed{\frac{1}{2} = \frac{1}{2}}$

$\log_b a = c$, por lo tanto se cumple el segundo caso.

$T(n) = \Theta(f(n) \lg n) = \Theta(\sqrt{n} \lg n)$

Ecuaciones de Recurrencia con Método Maestro (No simplificado):

B,C,E

Ecuaciones de Recurrencia con Método Maestro

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

$a = 2\ ,\ b = \frac{10}{7}\ ,\ F(n) = n$

$\log_b a = \log_{\frac{10}{7}} 2 = 1{,}94$

Verifiquemos si cumple con el caso 1:

$F(n) = O\left(n^{\log_b a - \epsilon}\right)$ $\qquad \epsilon > 0$

$n = O\left(n^{1{,}94 - \epsilon}\right)$

$n = O\left(n^{1{,}94 - 0{,}94}\right)$ $\qquad$ Si tomamos $\epsilon = 1{,}94$ cumple que sea $\epsilon > 0$

$n = O\left(n^{1}\right)$

$\boxed{n = O(n)}$

Se cumple el primer caso por lo tanto el $T(n)$ es:

$T(n) = \Theta\left(n^{\log_b a}\right) = \boxed{\Theta\left(n^{1{,}94}\right)}$

c.)

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

$a = 16$, $b = 4$, $F(n) = n^2$

$$\log_b a = \log_4 16 = \boxed{2}$$

Analizamos el segundo caso:

$$F(n) = \Theta(n^{\log_b a})$$

$$\boxed{n^2 = \Theta(n^2)}$$

Se cumple el segundo caso, por lo tanto el $T(n)$ es:

$$T(n) = \Theta(n^{\log_b a} \cdot \lg n) = \boxed{\Theta(n^2 \lg n)}$$

e.)

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

$a = 7$, $b = 2$, $F(n) = n^2$

$$\log_b a = \log_2 7 = \boxed{2.80}$$

Analizamos el primer caso:

$$F(n) = O(n^{\log_b a - e}) \qquad e > 0$$

$$n^2 = O(n^{2,80 - e})$$

$$n^2 = O(n^{2,80 - 0,80})$$

Tomamos $e = 0.80$ y vemos que es mayor que 0.

$$\boxed{n^2 = O(n^2)}$$

Se cumple el primer caso, por lo tanto $T(n)$ es:

$$T(n) = \Theta(n^{\log_b a}) = \boxed{\Theta(n^{2,80})}$$

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.**