Trabajo Práctico Complejidad

Ejercicio 1:

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

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_o tal que:

$$T(n) \le cf(n)$$
 cuando $n \ge n_0$

Si lo aplicamos al enunciado quedaría:

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

|--|

El algoritmo está explicado en el archivo.py

```
def middlesort(A):
    length = len(A)
    middlepos = math.trunc(length/2)
    middle = A[middlepos]
    minorspos = LinkedList()
    minorsposinverted = LinkedList()
    minorscounter = 0
    minorsmiddlecounter = 0
    for i in range(0,length):
        if A[i] < middle:</pre>
            if i < middlepos:</pre>
                minorsmiddlecounter +=1
            minorscounter += 1
            insert(minorspos,i,j)
            add(minorsposinverted,i)
    flag = math.trunc(minorscounter/2)
```

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.

```
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 [170,90,2,24,45,75] array queda asi:
```

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 [2,24,45,170,75,90]
 y queda asi:
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 [2,24,45,75,90,170]
 y queda asi:
```

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 $\Theta(n)$ y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para $n \le 2$. 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

Ejection 7)

Eurociones de recurrous con Nétado Maestro Simplipido

7.
$$\Gamma(n) = 2T(n/2) + n^4$$
 $a = 2$, $b = 2$, $c = 4$
 $\log_2 a = \log_2 2 = \left[\frac{1}{2} + 4\right]$

Como $c > \log_2 2$, esta ecusción se pude sooler con el prime case. Por la labor

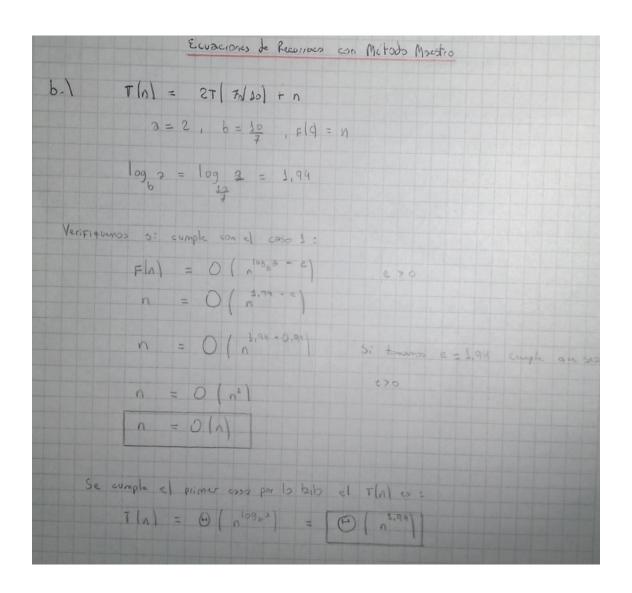
 $T(n) = O(r^{[n]}) = O(n^4)$
 $a = 7$
 $a =$

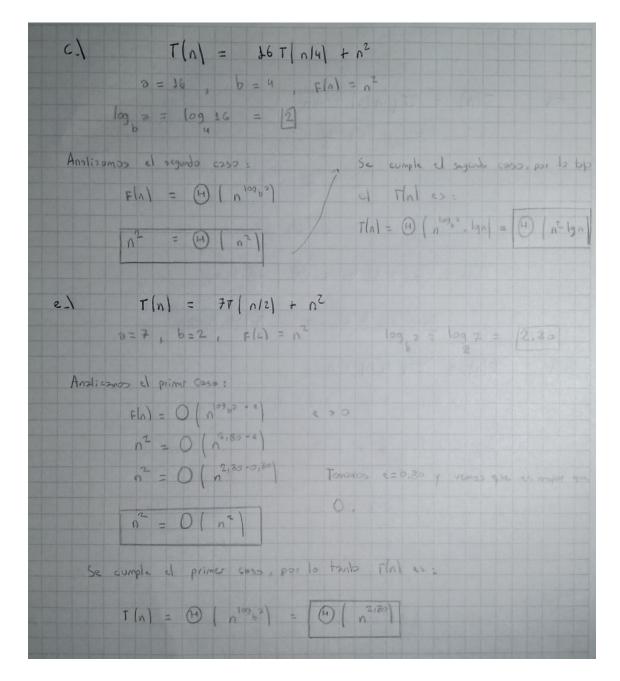
F.
$$T(n) = 2T(n/4) + \sqrt{n}$$
 $2 = 2$, $b = 4$, $c = \frac{1}{2}$
 $\log_{b} 2 = \log_{4} 2 = \left| \frac{1}{2} = \frac{1}{2} \right|$
 $\log_{b} 2 = C$, por lo tanto se cumple el segúndo caso.

 $T(n) = \Theta(|E(n)|| |g_{n}| = |\Theta|(|n||) |g_{n}| = |\Theta|(|n||) |g_{n}| = |\Theta|(|n||) |g_{n}| = |G|(|n||) |g_$

Ecuaciones de Recurrencia con Método Maestro (No simplificado):

B,C,E





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.