UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

ALGORITMOS Y ESTRUCTURAS DE DATOS II

Análisis de Complejidad por casos

Yacante Daniel

Leg: 10341

Ejercicio 1:

Demuestre que 6n3 ≠ O(n2).

$$T(6n^3) = T(n^3)$$

Nuestra función tiene orden de $\boldsymbol{n^3}$, por lo que no puede tener un peor caso de $O(n^2)$

Ejercicio 2:

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

El array debería estar ordenado ya que de esta forma cualquier pivote que se elija siempre está en la posición que le corresponde y no hay que buscar los elementos menores y mayores al pivote.

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: $log\ n$
- MergeSort: $n\ log\ n$
- InsertionSort: n

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.

```
"""
Ejercicio 4
Primero ordeno la lista con alguno de los métodos, aquí uso el método sort() para simplicidad.
Una vez ordenado calculo 2 indices necesarios, el del medio de la lista, y el del cuarto de la lista, ya que
del lado izquierdo tienen que quedar la mitad de los menores al de la mitad.
Luego lo que hago es simplemente intercambiar los elementos que están entre la mitad y el cuarto con la misma
cantidad de elementos del lado derecho.
"""

def ordenarMitad(A:list):
    Asorted=[]
    Asorted=A.copy()
    Asorted.sort()
    mid=round((len(Asorted)-1)/2)
    quad=round((mid)/2)
    for i in range(1,mid-quad+1):
        Asorted[mid-i], Asorted[mid+i] = Asorted[mid+i], Asorted[mid-i]
    return Asorted
```

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
"""
Ejercicio 5
Primero se ordena la lista, se puede hacer con cualquier método visto, aquí use el método sort()
por simplicidad del código. Por lo que la elección del método de ordenamiento tendrá impacto en el
orden de complejidad total.
Una vez ordenada la lista, me fijo si la suma de los elementos de los extremos son iguales al numero solicitado
Si es mayor, me muevo una posición hacia la izquierda del lado mayor, asi la suma siguiente tendría que ser menor
Si es menor, me muevo una posición hacia la derecha del lado menor, asi la suma siguiente tendría que ser mayor
Hasta llegar a que ambas posiciones son la misma y si se llega a ese punto es que no existe un par que de la suma.
"""
def ContieneSuma(A:list,n:int):
    Asorted=[]
    Asorted=A.copy()
    Asorted.sort()
    i=0
    d=len(Asorted)-1
    HS=False
    while i!=d:
        suma=Asorted[i]+Asorted[d]
        if suma==n:
            HS=True
            break
        elif suma<n:
            i+=1
        else:
            d-=1

    return HS
```

El orden de complejidad sería el orden del método de ordenamiento elegido mas n ya que la lista se recorrerá en el peor de los casos en su totalidad. $O(n + O(ordenamiento))$.

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:

En este método se asume que la entrada tiene una distribución uniforme de sus valores. El algoritmo el rango de los datos de entrada en n subintervalos de igual tamaño, siendo n el tamaño de la entrada. En cada subintervalo se colocaran los elementos del array original en otros array pero ahora cada elemento del subarray pertenecerá al mismo intervalo de rangos. Estos subarray se ordenarán con el método insertSort cada vez que se incorpore un nuevo elemento. Cuando no quedan mas elementos para ingresar a los subintervalos, el método de ordenamiento consiste en tomar cada subintervalo y colocarlos uno detrás de otro ya que estarán ordenados previamente.

Como ejemplo tomo el dado en la pagina 201 de “Introduction to Algorithms 3rd edition” de The MIT Press.

	A
1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68

Tomando como array de entrada A al de la figura. Primero se divide al rango de datos entre [0,1) en 10 partes iguales, ya que el array tiene una longitud de 10.

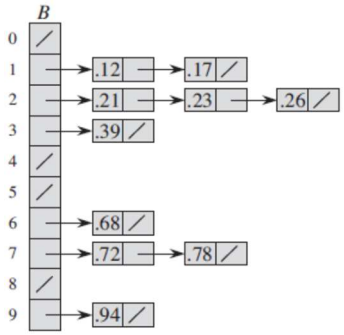
Por lo que los intervalos serán:

$$[0,0.1), [0.1,0.2), [0.2,0.3), [0.3,0.4), [0.4,0.5), [0.5,0.6), [0.6,0.7), [0.7,0.8), [0.8,0.9), [0.9,1)$$

El próximo paso es generar un nuevo array B de longitud 10 también, en el cual para cada posición se colocara un array que contendrá los elementos de A correspondientes a cada uno de los intervalos previamente generados.

El primer paso sería tomar el primer elemento de A, y colocarlo en la posición en B que le corresponda, en este caso será en B[7], luego tomamos el segundo elemento de A y lo colocamos en la posición de B que corresponda, y así con cada uno de los elementos de A.

Luego de que todos se han pasado por todos los elementos de A se han colocado en B, queda una figura como la siguiente.



El último paso que quedaría seria el de concatenar cada una de las listas que hay en B para obtener la lista A ordenada de forma creciente.

El orden de complejidad para el peor caso seria que la lista no se encuentre uniformemente distribuida y todos los elementos queden dentro del mismo subintervalo, por lo que este método se verá altamente dependiente del método de ordenamiento que elijamos para las subarray. En el caso promedio tendremos que recorrer el vector A se hace en O(n), el ordenamiento de las sublistas dependiendo del método se puede hacer en $n \cdot O\left(2 - \frac{1}{n}\right)$ (demostración en el mismo libro) y la concatenación se hacen en O(n), por lo que el caso promedio tiene complejidad de $\Theta(n)$.

Ejercicio 7

a -

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

$a = 2$, $b = 2$, $f(n) = n^4$ $\Big\}$ $n^{\log_2(2)} = n \Rightarrow \Theta(n)$

- casos 1 y 2 descartados ya que $n < n^4$
- caso 3:

$n^{\log_2(2)+\epsilon} \rightarrow$ si $\epsilon = 3 \Rightarrow \Omega\left(n^{\log_2 2 + 3}\right) > \Theta(n)$

verificación

$af(n/b) = 2f\left(\frac{n}{2}\right) = 2\left(\left(\frac{n}{2}\right)^4\right) = 2\frac{n^4}{2^4} = \frac{n^4}{2^3}$

Si $c = \frac{1}{2^3} \Rightarrow af(n/b) \leq cf(n)$

por lo tanto:

$T(n) = \Theta(f(n)) = \Theta(n^4)$

b -

$T(n) = 2T\left(\frac{7n}{10}\right) + n$

$a = 2$, $b = \frac{10}{7}$ $\Big\}$ $n^{\log_{\frac{10}{7}} 2} \approx n^{1,943}$

$f(n) = n$

caso 1:

$n^{\log_{\frac{10}{7}} 2 - \epsilon}$, existe un $\epsilon \approx 0,943$ tal que $n^{\log_{\frac{10}{7}} 2 - \epsilon} = f(n)$

por lo tanto

$T(n) = \Theta(n^{1,943})$

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

$a = 16$, $b = 4$, $f(n) = n^2$ $\Big\}$ $n^{\log_4 16} = n^2$

caso 2:

$n^{\log_4 16} = n^2 = f(n) \Rightarrow T(n) = \Theta(n^2 \cdot \log n)$

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

$a = 7$, $b = 3$, $c = 2$ $\Big\}$ $\log_3(7) \cong 1{,}77 < c \Rightarrow$ caso 3: $T(n) = \Theta(n^2)$

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

$a = 7$, $b = 2$, $c = 2$ $\Big\}$ $\log_2 7 \cong 2{,}8 > c \Rightarrow$ caso 1: $T(n) = \Theta(n^{2{,}8})$

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

$a = 2$, $b = 4$, $c = 0{,}5$ $\Big\}$ $\log_4 2 = 0{,}5 = c \Rightarrow$ caso 2: $T(n) = (\sqrt{n} \log n)$