Algoritmos y Estructuras de Datos II

Ejercitación: Análisis de Complejidad por casos

Ejercicio 1:

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

Consideramos $6n^3$ como f(n) y a n^2 como c.g(n) (siendo c=1)

$6n^3 \leq n^2$

$6n.n^2 \leq 1.n^2$

$6n.n^2 \leq 1.n^2$

Por inducción:

n=1:

$6.(1).(1)^2 \leq 1.(1)^2$

$6.1 \leq 1.1$

$6 \leq 1$ => **Falso**

En conclusión, $6n^3 \neq O(n^2)$

Ejercicio 2:

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

Una lista de la forma:

[10, 9, 7, 8, 6, 3, 4, 2, 1, 5]

Tomando como pivote siempre el medio (el primer caso sería **5**)

Complejidad: O(nlogn)

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: $O(n^2)$

Insertion-Sort: O(n)

Merge-Sort: O(nlogn)

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 entrada

7	3	2	8	5	4	1	6	10	9

Link Repl.it: https://replit.com/@Paulonia/complejidad#main.py

```python
def MidSort(L):
    pivotpos=round(len(L)/2)
    pivot=L[pivotpos]
    menores=0
    mayores=0
    flag=False
    for i in range(0,pivotpos):
        if L[i]<pivot:
            menores+=1
        else:
            mayores+=1
    #Caso 0: La Lista ya está ordenada
    if menores==math.trunc(pivotpos/2):
        print("Lista Ordenada")
        return L
    #Caso 1: Hay más números mayores que menores al pivote en la parte izquierda de la lista
    if mayores>menores:
        for i in range(0,len(L)):
            if L[i]>pivot and i<pivotpos:
                numMayor=L[i]
                posMayor=i
            if L[i]<pivot and i>pivotpos:
                numMenor=L[i]
                posMenor=i
                flag=True
        if flag==True:
            #Reemplazo un número menor al pivote de la parte derecha en un número mayor al pivote en la parte izquierda
            L[posMayor]=numMenor
            L[posMenor]=numMayor
            return MidSort(L)
        else:
            #El pivote tiene todos números mayores a la derecha, por lo que no se puede reemplazar y quedaría así la lista
            return L
    #Caso 2: Hay mas números menores que mayores al pivote en la parte izquierda de la lista
    if menores>mayores:
        for i in range(0,len(L)):
            if L[i]<pivot and i<pivotpos:
                numMenor=L[i]
                posMenor=i
            if L[i]>pivot and i>pivotpos:
                numMayor=L[i]
                posMayor=i
                flag=True
        if flag==True:
            #Reemplazo un número mayor al pivote de la parte derecha en un número menor al pivote en la parte izquierda
            L[posMenor]=numMayor
            L[posMayor]=numMenor
            return MidSort(L)
        else:
            #El pivote tiene todos números menores a la derecha, por lo que no se puede reemplazar y quedaría así la lista
            return L
    return L
```

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)):
        for j in range (0,len(A)):
            #Suma los elementos de la lista (que no sean el mismo) y los compara con n
            if A[i]+A[j]==n and i!=j:
                return True
    return False
```

El costo computacional es O(n^2)

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.

Bucket Sort:

Este algoritmo de ordenamiento consiste en crear *buckets* o "baldes"(los cuales serían como cajas o "secciones", pero representados en listas) para distribuir los elementos de la lista allí. Si tenemos de 1 a 10 elementos, entonces vamos a crear un array de 10 elementos los cuales van a ser nuestros buckets.

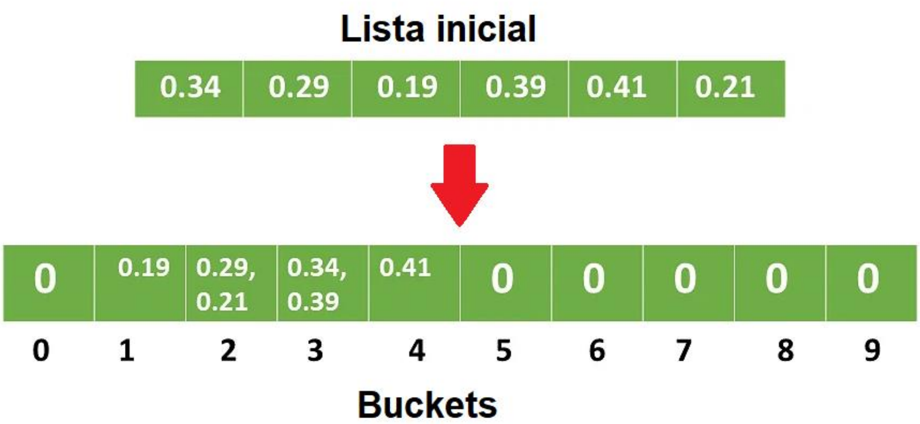
La inserción en los *buckets* se hace con la siguiente ecuación=

(Valor actual de la lista * número de elementos de la lista (n)) / valor máximo de elementos + 1

al resultado lo truncamos y nos dará, por ejemplo, 4. Entonces tenemos que colocar ese elemento de la lista en el *bucket* número 4 (posición 5 del array, ya que va de 0 a k-1). Cada elemento del array va a ser una lista, por lo que va a tener una complejidad espacial bastante alta.

Una vez que se colocan todos los elementos, se debe ordenar cada *bucket* por separado utilizando *Insertion Sort*, y luego se concatenan todos los *buckets* en orden en una lista, la cual sería la resultante.

<u>*Ejemplo*</u>:



(0.34*10)/1 = 3 (0.29*10)/1 = 2 (0.19*10)/1 = 1 (0.39*10)/1 = 3 (0.41*10)/1 = 4

(0.21*10)/1 = 2

Ordenamos los *buckets* y los concatenamos ordenados en una lista:

0	0.19	0.21, 0.29	0.34, 0.39	0.41	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

0.19	0.21	0.29	0.34	0.39	0.41

Y ya tenemos la lista ordenada.

Complejidad Temporal:
- Peor caso: $O(n^2)$
- Caso promedio: $O(n)$
- Mejor caso: $O(n+k)$

Siendo n el número de elementos y k el número de *buckets*.

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

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

$a = 2 \quad b = 2 \quad c = 4$

$\text{Log}_b a = \text{Log}_2 2 = 1 < 4 \; (c)$

$T(n) = O(n^c) = O(n^4)$ (Caso 3)

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

$a = 2 \quad b = 10/7 \quad c = 1$

$\text{Log}_b a = \text{Log}_{10/7} 2 \simeq 1{,}94 > 1 \; (c)$

$T(n) = O(n^{\log_b a}) = O(n^{1{,}94})$ (Caso 1)

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

$a = 16 \quad b = 4 \quad c = 2$

$\log_b a = \log_4 16 = 2 = 2\ (c)$

$T(n) = O(n^c \log n) = O(n^2 \log n)$ (Caso 2)

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

$a = 7 \quad b = 3 \quad c = 2$

$f(n) = n^2 = O(n^{\log_3 7 + \varepsilon})$

$n^2 = O(n^{1,77 + \varepsilon})$

$\varepsilon \simeq 0,23$

Caso 3: $a f(n/b) \leq c f(n) \quad (c < 1)$

$7(n/3)^2 \leq c n^2$

$7 \cdot \frac{n^2}{9} \leq c n^2 \rightarrow$ para $c = \frac{7}{9} < 1$

Por lo tanto, $T(n) = O(n^2)$

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

$f(n) = n^2 = O(n^{\log_2 7 + \varepsilon})$

$n^2 = O(n^{2,8 + \varepsilon})$

$2 - 2,8 = \varepsilon \simeq 0,8$ (Caso 1)

Por lo tanto, $T(n) = O(n^{\log_b a}) = O(n^{2,8})$

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

$a = 2 \quad b = 4 \quad c = 1/2$

$f(n) = n^{1/2} = O(n^{\log_4 2 + \varepsilon})$

$n^{1/2} = O(n^{1/2 + \varepsilon})$

Caso 2: $c = \log_b a$

Por lo tanto:

$T(n) = O(n^{\log_b a} \cdot \log n) = O(n^{1/2} \cdot \log n)$