Algoritmo y Estructuras de datos II

TP: Complejidad

Ejercicio 1: Demuestre que $6n^3 \neq O(n^2)$

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

$$T(n) \leq c \cdot f(n) \quad \text{para } n \geq n_0$$

por lo que debería cumplirse:

$$6n^3 \leq c \cdot n^2 \quad \text{para } n \geq n_0$$

Inducción

paso inicial $n=0$

$$6 \cdot 0^3 \leq c \cdot 0^2$$

$0 \leq 0$ se cumple en el paso inicial

para $n=k$

$6 \cdot k^3 \leq c \cdot k^2$ suponemos que se cumple

para $n=k+1$

$$6 \cdot (k+1)^3 \leq c \cdot (k+1)^2 = 6 \cdot \frac{(k+1)^3}{(k+1)^2} \leq c \cdot \frac{(k+1)^2}{(k+1)^2} =$$

$$6 \cdot (k+1) \leq c = 6k+6 \leq c$$

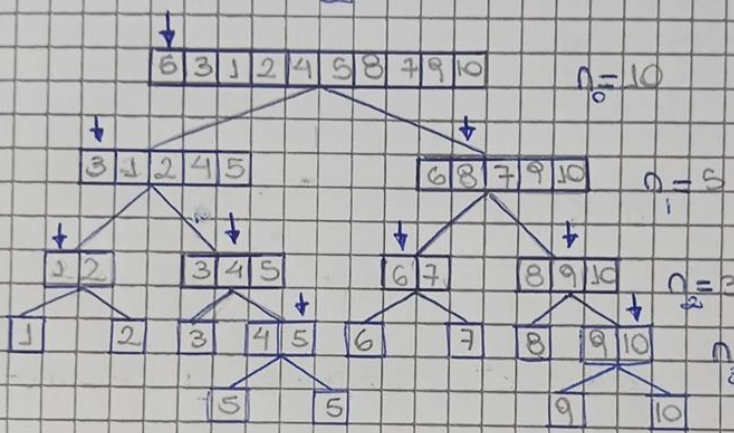
$$\lim_{k \to \infty} (6k+6) \leq \lim_{k \to \infty} c$$

llegamos a un absurdo por lo que

$$6n^3 \neq O(n^2)$$

Ejercicio 2. ¿Cómo sería un array de números (mínimo 10) para el mejor caso de la estrategia Quicksort(n)?

Para poder obtener el mejor caso de Quicksort es necesario hacer la elección de un pivote ideal. Considerando que poseemos n elementos, buscaríamos que el pivote divida la lista a la mitad en caso de que n sea par obtendríamos $n/2$ elementos en cada lista y en caso de ser impar obtendríamos una lista con $n/2$ elementos y otra con $n/2+1$ elementos.



Además, el pivote en la primer iteración debería encontrarse en la primera posición para no tener que recorrer la lista. Luego en las sublistas el próximo pivote debería encontrarse en la primera posición o en el caso que la sublista contenga el pivote anterior este debería encontrarse en la casilla siguiente (porque el pivote anterior siempre ocupaba un extremo).

Complejidad: La primer iteración recorre n en la segunda $n/2 + n/2 = n$ ...

$$\underbrace{n + \underbrace{n/2 + n/2}_{n} + \underbrace{n/4 + n/4 + n/4 + n/4}_{n} \cdots}_{c \cdot n} = n \cdot \log_2 n$$

$$O(c \cdot \log_2 n)$$

X-

TP Complejidad

Ejercicio 3: ¿Cuál es el tiempo de ejecución de la estrategia QuickSort(A), InsertionSort(A) y MergeSort(A) cuando todos los elementos del array A tienen el mismo valor?

QuickSort(A): Dependerá de la implementación que realicemos al comparar el pivote con el resto de elementos. Al comparar si utilizamos >= obtendríamos arrays del mismo tamaño que el original menos un elemento correspondiente al pivote. Este caso es similar al peor caso donde siempre el pivote es el mayor o menor elemento del array y su complejidad es de n^2.

InsertionSort(A): Si avanzamos en el array en la primer iteración realizando una comparación del mayor elemento respecto de los anteriores, podríamos recorrer el array de elementos iguales sin seleccionar ninguno para ser ordenado. En conclusión, su complejidad será de n.

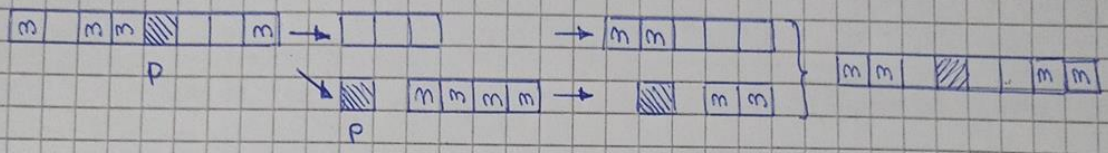
MergeSort(A): El proceso de dividir el array ocurriría de la misma forma ya que este no considera el contenido del Array. En el momento de unir el array devuelta procede a comparar, si implementamos una bandera que determine que el subarray contiene elementos iguales, podríamos unir cada subarray solo comparando la bandera y el primer elemento. Por lo que sería posible obtener una complejidad 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.

Ejemplo de la lista de salida

7	3	2	8	5	4	1	6	10	9
				↑					

Mi algoritmo busca el valor del medio, lo toma como pivote. Crea sublista para números menores al pivote, recorre la lista insertando los menores en la nueva sublista. Luego, divide esa divide esa sublista a la mitad y la reinserta de tal modo que mitad de sus elementos queden a cada lado de la lista.




```
def SortMind(L):

    n=len(L)
    pivot = acces(L, n/2)
    Lnew = Array(0,n/2)
    count=0

    for i in range(0, n):

        if L[i] < pivot:

            Lnew[count] = L[i]
            delete(L, L[i])
            count+=1
        delete(L, pivot)

    count2=0

    for i in range(0, count/2):

        insert(L, (n-count)/2, Lnew(0))
        delete(L, Lnew[0])
        count= count+1

    insert(L, (n-count)/2 + count2, pivot)

    for i in range(0, count - count2):

        insert(L, n-count + count2+1, Lnew[0])

        delete(Lnew, Lnew[0])
```

Ejercicio 5: Implementar un algoritmo `def Contiene_Suma(A,n)` que dados 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):
    Asize = len(A)
    for i in range(0, Asize):
        sum = A[i]
        for j in range(i, Asize):
            if sum + A[j] == n:
                return True
```

Complejidad:

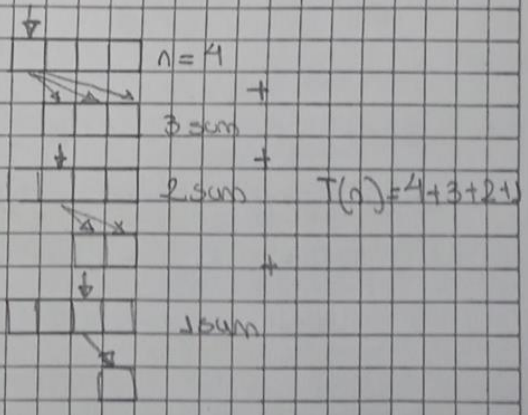
En el mejor caso el primer elemento sumado con el segundo daría n, lo cual costaría 1OE por los accesos y su complejidad sería $\Omega(1)$

En el caso promedio podríamos considerar un elemento a mitad de lista que se sume con otro en el cuarto restante del Array. En ese caso su complejidad estaría dada por $T(n) = \frac{n}{2} + \sum_{i=n/2+1}^{n/2} n$ siendo $n \cdot \log_2 n$ una cota superior $\Theta(n \log n)$

En el peor caso no encontraría una suma que de n o la encontraría sumando el penultimo elemento con el ultimo por lo que

$T(n) = \sum_{i=0}^{n} n$ cuya cota superior es n^2 por lo que

$O(n^2)$ es su peor caso.



Caso prom:

$T(n) = \frac{n}{2} + \sum_{i=n/2+1}^{n/2} n$

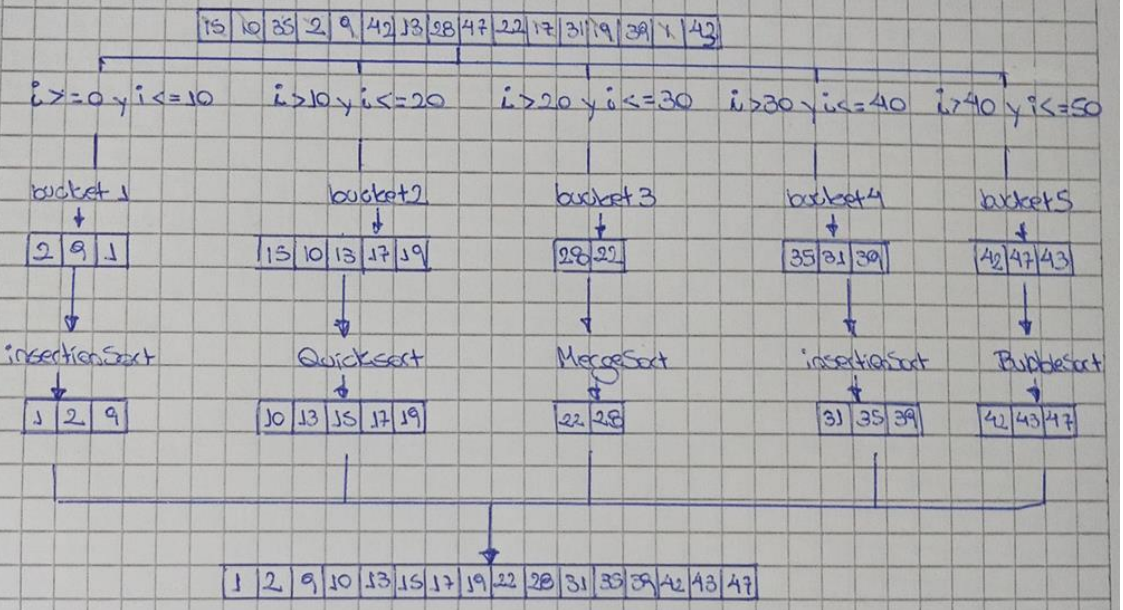
$T(4) = 2 + 3 + 2$

$T(n) = 7 \approx 8$

$n \cdot \log_2 n = 4 \cdot 2 = 8$

Ejercicio 6: Investigar otro algoritmo de ordenamiento como BucketSort, HeapSort o RadixSort, brindando un ejemplo que explique su funcionamiento en su caso promedio. Mencionar su orden y explicar sus casos promedio, mejor y peor.

BucketSort



Explicación: BucketSort crea casilleros, cada casillero posee una condición y todas las condiciones son excluyentes. Luego, reparte los elementos según su condición a lo largo de los casilleros, cada casillero posee su propio método de ordenamiento. Posteriormente los elementos son unidos de vuelta. Su mayor ventaja es que posee la capacidad de trabajar con hilos y dependiendo si las condiciones son equitativas y habiendo utilizado buenos ordenamientos en cada casillero, se considera que su caso promedio y mejor caso es $\Theta(n)$. Aun así, su peor caso es que las condiciones sean malas y los elementos se agrupen en un solo casillero con un método de ordenamiento es $O(n^2)$.