Trabajo Practico N°2: Complejidad Algorítmica

1) Demuestre que $6n^3 \neq \mathbf{O}(n^2)$

Sea una función f(n), decimos que f(n) pertenece a la familia de funciones $\Omega(g(n))$ si existe una constante C1 y un numero n0 tales que:

$0 \leq f(n) \leq C1 * g(n),$ para $n \geq n_0$

Entonces

$0 \leq 6n^3 \leq C1 * n^2$

$0 \leq 6n \leq C1$

Concluimos que no existe una constante C1 y un numero n0 tales que cumplan esa condición. Entonces $6n^3$ no pertenece a la familia de $O(n^2)$ (no la acota por arriba).

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

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

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(A) : n^2

Insertion-Sort(A) : n^2

Merge-Sort(A) : n*(log n)

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.

Explicacion: Ordenar la lista con algún algoritmo de Ordenamiento Avanzado (por ej mergesort) y simplemente buscaremos el 3er elemento ya que tendrá dos menores y lo que haremos será mover uno al final de la lista y al 3 a la mitad. Esto siempre será verdadero si la lista esta ordenada y tiene longitud n mayor a 3 podremos usar este algoritmo.

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.

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.

   BuckeSort: algoritmo de ordenamiento que ordena los elementos de una lisita dividiéndolos en botes dependiendo de su rango y luego ordena cada bote individualmente, esto último lo hace utilizando otro algoritmo de ordenamiento, como InsertionSort o MergeSort, o recursivamente utilizando el propio BucketSort. La eficacia del algoritmo de ordenamiento por botes está influenciada por diversos factores, como la cantidad de botes, la disposición de los elementos y el método de ordenamiento empleado para clasificar los botes

individualmente. En términos generales, su complejidad temporal promedio es de O(n + k), donde "n" es la cantidad de elementos y "k" representa la cantidad de botes.

HeapSort: Es un ordenamiento por montículos (heap en inglés), no recursivo, no estable, con complejidad Θ(n*log n), incluso en los peores casos.

Un Heap es un árbol binario completo a izquierda, que permite implementar una cola con prioridad, y donde los elementos se almacenan cumpliendo la propiedad de que la clave de un nodo siempre es mayor que la clave de cualquiera de sus hijos. Lo que nos asegura que la raíz del árbol, en un Heap, siempre es el elemento mayor de la estructura.

Un Heap tiene la propiedad de poder ser almacenado en un vector sin ocasionar inconvenientes en el manejo de la estructura, evitando el uso de punteros. El Heap se almacena en un arreglo A[1..n], y utiliza una variable m que indica cuantos elementos tiene el Heap.

RadixSort: Este algoritmo funciona trasladando los elementos a una cola, comenzando con el dígito menos significativo del número (El número colocado más a la derecha tomando en cuenta unidades, decenas, centenas, etc.). Cuando todos los elementos son ingresados a las colas, éstas se recorren en orden para después agregarlos al vector.

Este algoritmo es muy rápido en comparación con otros algoritmos de ordenación, sin embargo, ocupa más espacio de memoria al utilizar colas

O(n + k), caso promedio siendo n el numero de elementos a ordenar y k el rango de los valores posibles de los elementos.

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

a = 2, b = 2, f(n) = n^4

$\Theta(n^{\log_2 2}) = \Theta(n)$

Caso 1: $n^{\log_2 2 - e}$, con e>0

$n^{1-e} = n^4$, no hay e posible que cumpla esas condiciones

Caso 2: $n^{\log_2 2}$

$n^1 = n^4$, son de distinto orden

Caso 3: $n^{\log_2 2 + e}$, con e>0, comprobando que $af(n/b) \leq cf(n)$ para alguna constante c>1

$n^{1+e} = n^4$, con e=3 se cumple

Se cumple, pero tenemos que probar la segunda condición

af(n/b) = 3*f(n/4)

= 3 * (n^4/256)

= 3 * (n^4/256)≤ (3/256) * n^4

Consecuentemente, por el caso 3, T(n) = Θ(f(n)) = **Θ(n^4)**

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

a = 2, b =10/7, f(n) = n

$\Theta(n^{\log_{10/7}2}) \approx \Theta(n^{1,94})$

Caso 1: $n^{\log_{10/7}2 - e}$, con e>0

$n^{1,94 - e} = n$, se cumple con e = 0,94

Consecuentemente, por el caso 1, T(n) = $\Theta(n^{\log_b a})$ = **$\Theta(n^{1,94})$**

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

a = 16, b = 4, f(n) = n^2

$\Theta(n^{\log_4 16}) = \Theta(n^2)$

Caso 1: $n^{\log_4 16 - e}$, con e>0

$n^{2 - e} = n^2$, no hay e>0 que satisfaga esta ecuación.

Caso 2: $n^{\log_4 16}$

$n^2 = n^2$, son iguales

Consecuentemente, por el caso 2, T(n) = $\Theta(n^{\log_b a}) = \Theta(f(n))$ = **$\Theta(n^2)$**

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

a = 7, b = 3, c = 2

Caso 1: $\log_b a$ > c, T(n) = $\Theta(n^{\log_b a})$

1,77>2, no se cumple la condición.

Caso 2: $\log_b a$ = c, T(n) = Θ(f(n))*log n = $\Theta(n^c * \log n)$

1,77 = 2, es falso no se cumple la condición.

Caso 3: $\log_b a$ < c, T(n) = Θ(f(n)) = $\Theta(n^c)$

1,77<2, es Verdadero. Consecuentemente por el caso 3 T(n) = **Θ(n^2)**

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

a = 7, b = 2, c=2

Caso 1: $\log_b a$ > c, T(n) = $\Theta(n^{\log_b a})$

2,80 > 2, es Verdadero. Consecuentemente, por el caso 1, T(n) = $\Theta(n^{\log_b a})$ = **$\Theta(n^{2,80})$**

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

a = 2, b=4, c = ½

Caso 1: $\log_b a$ > c, T(n) = $\Theta(n^{\log_b a})$

½ > ½, es Falso. No se cumple la condición.

Caso 2: $\log_b a$ = c, T(n) = Θ(f(n))*log n = $\Theta(n^c * \log n)$

½ = ½, es Verdadero. Consecuentemente, por el caso 2,

T(n) = Θ(f(n))*log n = $\Theta(n^c * \log n)$= **$\Theta(n^{1/2} * \log n)$**