Tp – complejidad

Giuliano La Piano

Ejercicio 1:

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

T(n) es O(f(n)) si existen constantes positivas c y n0 tal que:
$T(n) \leq cf(n)$ cuando $n \geq n0$
$6n^3 \leq cn^2$ cuando $n \geq 0$, no se cumple para todo n ya que no existe una constante c que cumpla la desigualdad para todo $n \geq 0$.
$6n^3 \leq 7n^3$ cuando $n \geq 0$, se cumple para todo n. Por lo tanto $6n^3$ es de $O(n^3) \neq \boldsymbol{O}(n^2)$.

Ejercicio 2:

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

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?

Insertion-Sort(A): O(n)
Merge-Sort(A): O(nlogn)
QuickSort(A): $O(n^2)$

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

7	3	2	8	5	4	1	6	10	9

```python
def encuentra_elemento_medio(lista):
    if len(lista) == 1:
        return lista[0]
    else:
        medio = len(lista) // 2
        if lista[medio] == len(lista[:medio]):
            return lista[medio]
        elif lista[medio] > len(lista[:medio]):
            return encuentra_elemento_medio(lista[:medio])
        else:
            return encuentra_elemento_medio(lista[medio+1:])
```

```python
def ordena_lista(lista):
    if len(lista) <= 1:
        return lista
    else:
        elemento_medio = encuentra_elemento_medio(lista)
        menores = []
        mayores = []
        for elemento in lista:
            if elemento < elemento_medio:
                menores.append(elemento)
            elif elemento > elemento_medio:
                mayores.append(elemento)
        return ordena_lista(menores) + [elemento_medio] + ordena_lista(mayores)
```

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
from mylinkedlist import *

#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.

vector = [1,2,3,4]
print(len(vector))
print(vector)

def contienesuma(A,n):
  cont = 1
  for i in range(len(A)-1):
    for j in range(cont,len(A)):
      if A[i]+(A[j]) == n:
        return True
    cont += 1
  return False

print(contienesuma(vector,20))
```

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.

El algoritmo de ordenamiento radix sort se basa en la idea de ir ordenando los elementos de una lista de forma incremental, basándose en los valores de sus dígitos desde el dígito menos significativo hasta el más significativo. Este algoritmo es muy eficiente y puede ordenar grandes cantidades de datos en un tiempo muy reducido.

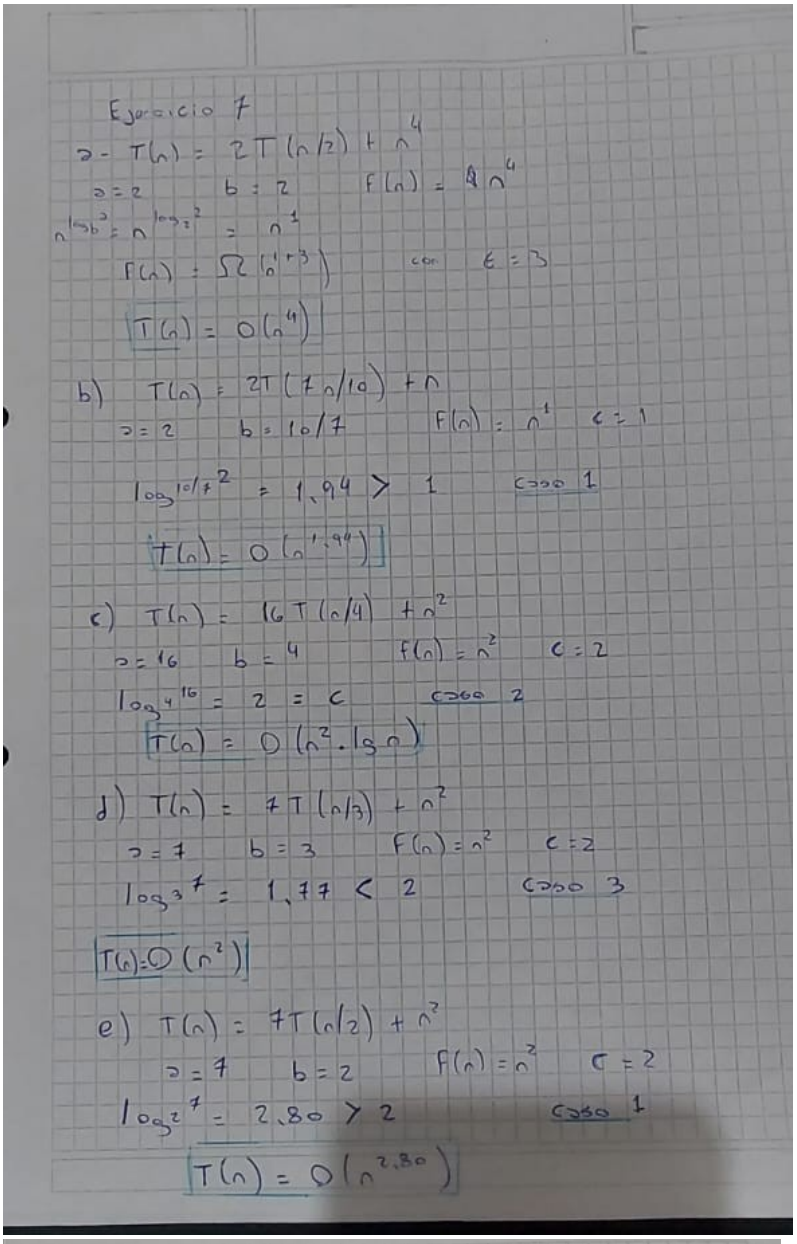
La complejidad del algoritmo radix sort depende del tamaño de los números a ordenar y del tamaño de la base del sistema numérico utilizado.
En el peor caso, la complejidad de tiempo de radix sort es O(d * (n + b)), donde d es el número de dígitos en el número más grande, n es el número de elementos a ordenar y b es la base del sistema numérico utilizado.

Esto significa que la complejidad de tiempo de radix sort es lineal en el número de elementos a ordenar (n), pero es proporcional al número de dígitos en el número más grande (d) y al tamaño de la base del sistema numérico utilizado (b).

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



Ejercicio 7

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

$a = 2 \quad b = 2 \quad F(n) = 9n^4$

$n^{\log_b a} = n^{\log_2 2} = n^1$

$F(n) = \Omega(n^{1+3})$ con $\epsilon = 3$

$\boxed{T(n) = O(n^4)}$

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

$a = 2 \quad b = 10/7 \quad F(n) = n^1 \quad c = 1$

$\log_{10/7} 2 = 1,94 > 1$ caso 1

$\boxed{T(n) = O(n^{1,94})}$

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

$a = 16 \quad b = 4 \quad F(n) = n^2 \quad c = 2$

$\log_4 16 = 2 = c$ caso 2

$\boxed{T(n) = O(n^2 . \lg n)}$

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

$a = 7 \quad b = 3 \quad F(n) = n^2 \quad c = 2$

$\log_3 7 = 1,77 < 2$ caso 3

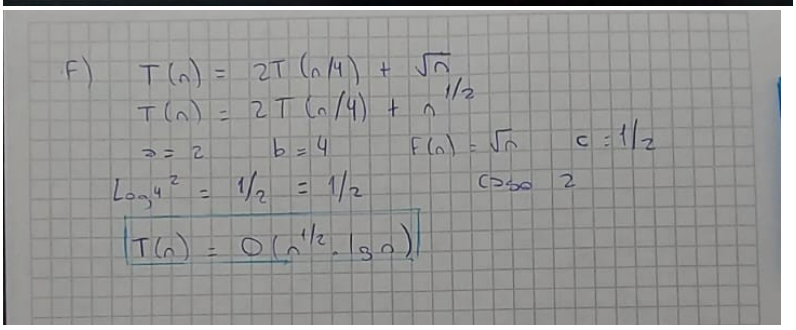
$\boxed{T(n) = O(n^2)}$

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

$a = 7 \quad b = 2 \quad F(n) = n^2 \quad c = 2$

$\log_2 7 = 2,80 > 2$ caso 1

$\boxed{T(n) = O(n^{2,80})}$



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

$T(n) = 2T(n/4) + n^{1/2}$

$a = 2 \quad b = 4 \quad F(n) = \sqrt{n} \quad c = 1/2$

$\log_4 2 = 1/2 = 1/2$ caso 2

$\boxed{T(n) = O(n^{1/2} . \lg n)}$