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) ?

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

Tomando como pivot el primer elemento de la lista. O(n log 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?

QuickSort(A): $O(n^2)$

Insertion-Sort(A): O(n)

Merge-Sort(A): 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 salida

7	3	2	8	5	4	1	6	10	9

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.

El algoritmo utilizado tiene un costo de O(nlogn) ya que es el costo de ordenar y recorrer la lista.

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.

RadixSort es un algoritmo de ordenamiento que ordena los elementos analizando cada uno dígito a dígito, desde el menos significativo hasta el más significativo.

El proceso de clasificación es el siguiente:

- Encontrar el elemento máximo y adquirir el número de dígitos de ese elemento. Nos da el número de iteraciones que seguirá el proceso de clasificación.
- Agrupe los dígitos individuales de los elementos en la misma posición significativa en cada iteración.
- El proceso de agrupación comenzará desde el dígito menos significativo y finalizará en el dígito más significativo.
- Ordenar los elementos según los dígitos en esa posición significativa.
- Mantener el orden relativo de los elementos que tienen el mismo valor clave. Esta propiedad del tipo de base lo convierte en un tipo estable.

Funcionamiento(ejemplo):

Para este ejemplo el input será la siguiente lista de números enteros:

[162,248,415,623,825]

Vamos a ordenar la lista en orden ascendente.

Paso 1: Identifique el elemento con el valor máximo en la lista. En este caso es 835.

Paso 2: Calcula el número de dígitos del elemento máximo. 835 tiene 3 dígitos exactamente.

Paso 3: Determine el número de iteraciones según el paso 2. 835 tiene 3 dígitos, lo que significa que el número de iteraciones será 3.

Paso 4: Determinar la base de los elementos. Como se trata de un sistema decimal, la base será 10.

Paso 5: Iniciar la primera iteración

Primera iteración:

En la primera iteración, consideramos el valor posicional unitario de cada elemento.

Paso 1: Modifica el número entero en 10 para obtener el lugar unitario de los elementos. Por ejemplo, 623 mod 10 nos da el valor 3 y 248 mod 10 nos da 8.

Paso 2: Utilice la clasificación por conteo o cualquier otra clasificación estable para organizar los números enteros según su dígito menos significativo. Como se ve en la figura, 248 caerán en el octavo cubo. 8 caerá en el tercer cubo y así sucesivamente.

Después de la primera iteración, la lista ahora tiene este aspecto.

[162,623,835,415,248]

Segunda iteración:

En esta iteración, consideraremos el dígito en el 10th lugar para el proceso de clasificación.

Paso 1) Divide los números enteros entre 10. 248 dividido entre 10 nos da 24.

Paso 2: Modifique la salida del paso 1 por 10. 24 mod 10 nos da 4.

Paso 3: Siga el paso 2 de la iteración anterior.

Después de la segunda iteración, la lista ahora se ve así

[415,623,835,248,162]

Tercera iteración:

Para la iteración final, queremos obtener el dígito más significativo. En este caso son los 100th lugar para cada uno de los números enteros de la lista.

Paso 1: Dividimos los números enteros entre 100… 415 dividido entre 100 nos da 4.

Paso 2: Modifica el resultado del paso 1 por 10. 4 mod 10 nos da 4 nuevamente.

Paso 3: Siga el paso 3 de la iteración anterior.

La lista ahora se ve así:

[162,248,415,623,835]

Como podemos ver, la lista ahora está ordenada en orden ascendente. La iteración final se completó y el proceso de clasificación ya finalizó.

Complejidad Temporal:

Tiene una complejidad temporal de O(d * (n + b)), donde d es el número de dígitos, n es el número de elementos y b es la base del sistema numérico que se está utilizando. La complejidad temporal es la misma para el mejor, peor y promedio de los casos.

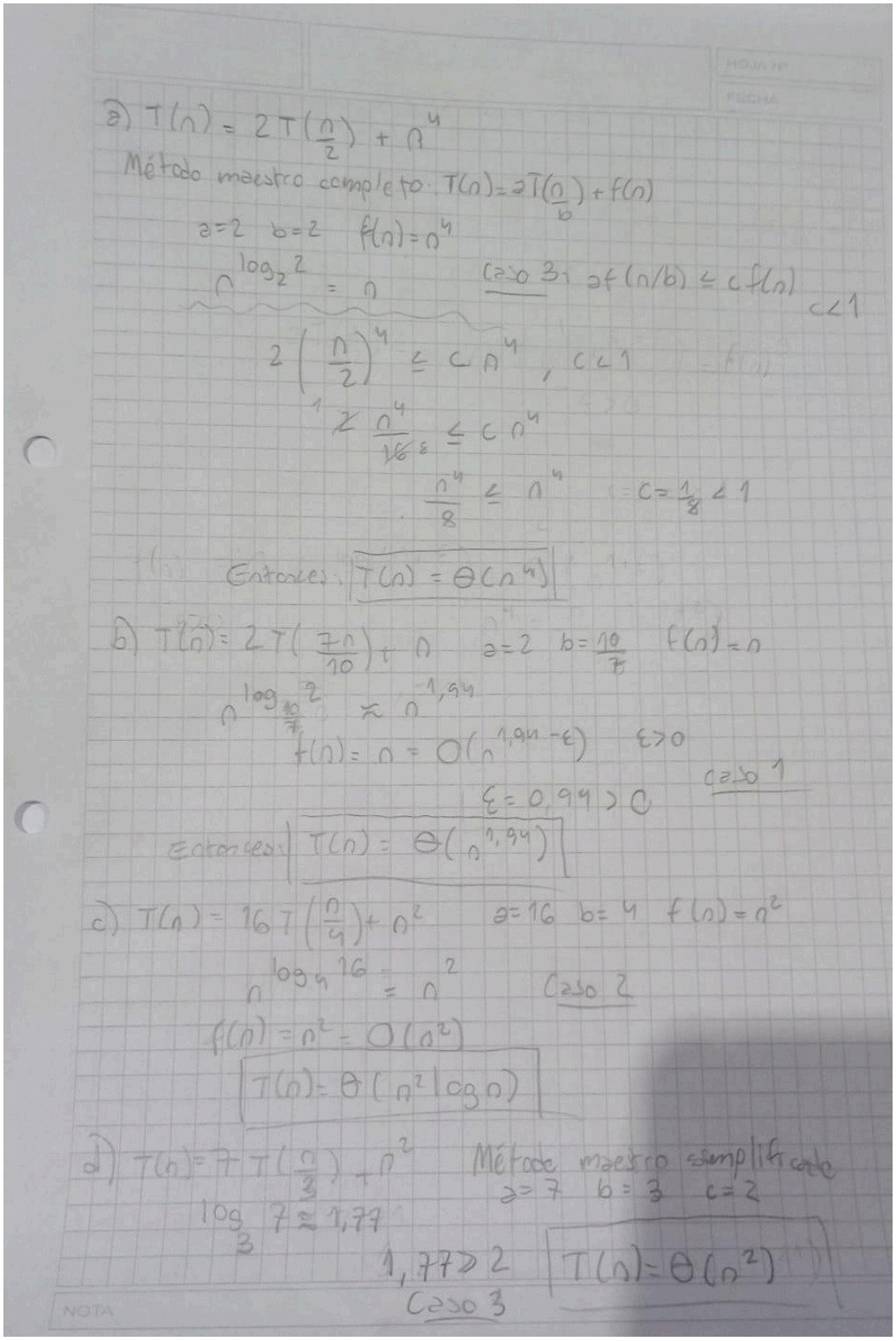
En implementaciones prácticas, radix sort a menudo es más rápido que otros algoritmos de ordenamiento basados en comparaciones, como quicksort o merge sort, para conjuntos de datos grandes, especialmente cuando las claves tienen muchos dígitos. Sin embargo, su complejidad temporal crece linealmente con el número de dígitos, por lo que no es tan eficiente para conjuntos de datos pequeños.

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



a) $T(n) = 2T(\frac{n}{2}) + n^4$

Método maestro completo: $T(n) = aT(\frac{n}{b}) + f(n)$

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

$n^{\log_2 2} = n$ Caso 3: $af(n/b) \leq cf(n) \quad c<1$

$2\left(\frac{n}{2}\right)^4 \leq cn^4, \quad c<1$

$2\frac{n^4}{16} \leq cn^4$

$\frac{n^4}{8} \leq n^4 \quad c=\frac{1}{8}<1$

Entonces: $\boxed{T(n) = \Theta(n^4)}$

b) $T(n) = 2T(\frac{7n}{10}) + n \quad a=2 \quad b=\frac{10}{7} \quad f(n)=n$

$n^{\log_{\frac{10}{7}} 2} \approx n^{1,94}$

$f(n) = n = O(n^{1,94-\varepsilon}) \quad \varepsilon>0$

$\varepsilon = 0,94 > 0$ Caso 1

Entonces: $\boxed{T(n) = \Theta(n^{1,94})}$

c) $T(n) = 16T(\frac{n}{4}) + n^2 \quad a=16 \quad b=4 \quad f(n)=n^2$

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

$f(n) = n^2 = O(n^2)$

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

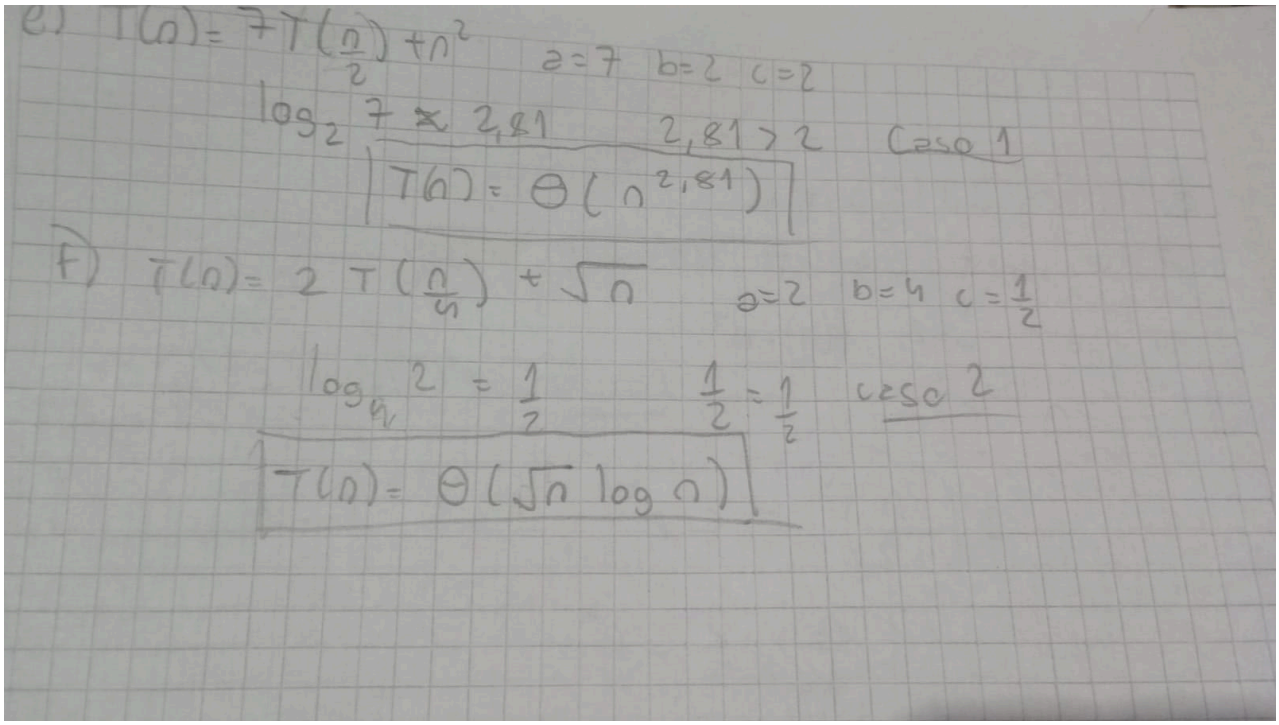
d) $T(n) = 7T(\frac{n}{3}) + n^2$ Método maestro simplificado

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

$\log_3 7 \approx 1,77$

$1,77 > 2$ Caso 3 $\boxed{T(n) = \Theta(n^2)}$

NOTA


e) $T(n) = 7T(\frac{n}{2}) + n^2$ $a=7$ $b=2$ $c=2$
$\log_2 7 \approx 2,81$ $2,81 > 2$ Caso 1
$T(n) = \Theta(n^{2,81})$
f) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$ $a=2$ $b=4$ $c=\frac{1}{2}$
$\log_4 2 = \frac{1}{2}$ $\frac{1}{2} = \frac{1}{2}$ caso 2
$T(n) = \Theta(\sqrt{n} \log n)$