**Francisco Alba**

**Ejercicio 1:**

Demuestre que 6n3 ≠ ***O***(n2).

T(n) es O(f(n)) si existen constantes c y n0 positivas, tales que T(n) ≤ cf(n) para todo n ≥ n0.

Considerando T(n) = 6n3, podemos observar que incluso para alguna constante c muy grande, existe un n = n0 a partir del cual 6n3 ≥ cn2 para todo n. Luego 6n3 ≠ O(n2).

**Ejercicio 2:**

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

Tomando como pivote siempre el elemento del medio del arreglo, la siguiente es una posibilidad:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Al estar el arreglo ordenado en forma creciente, en cada división sucesiva la parte izquierda y la derecha se matienen equilibradas y entonces el algoritmo logra ejecutarse en 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?

Cuando todos los elementos del arreglo tienen el mismo valor, los tiempos de ejecución son los siguientes:

QuickSort: O(n2)

InsertionSort: O(n)

MergeSort: (nlogn)

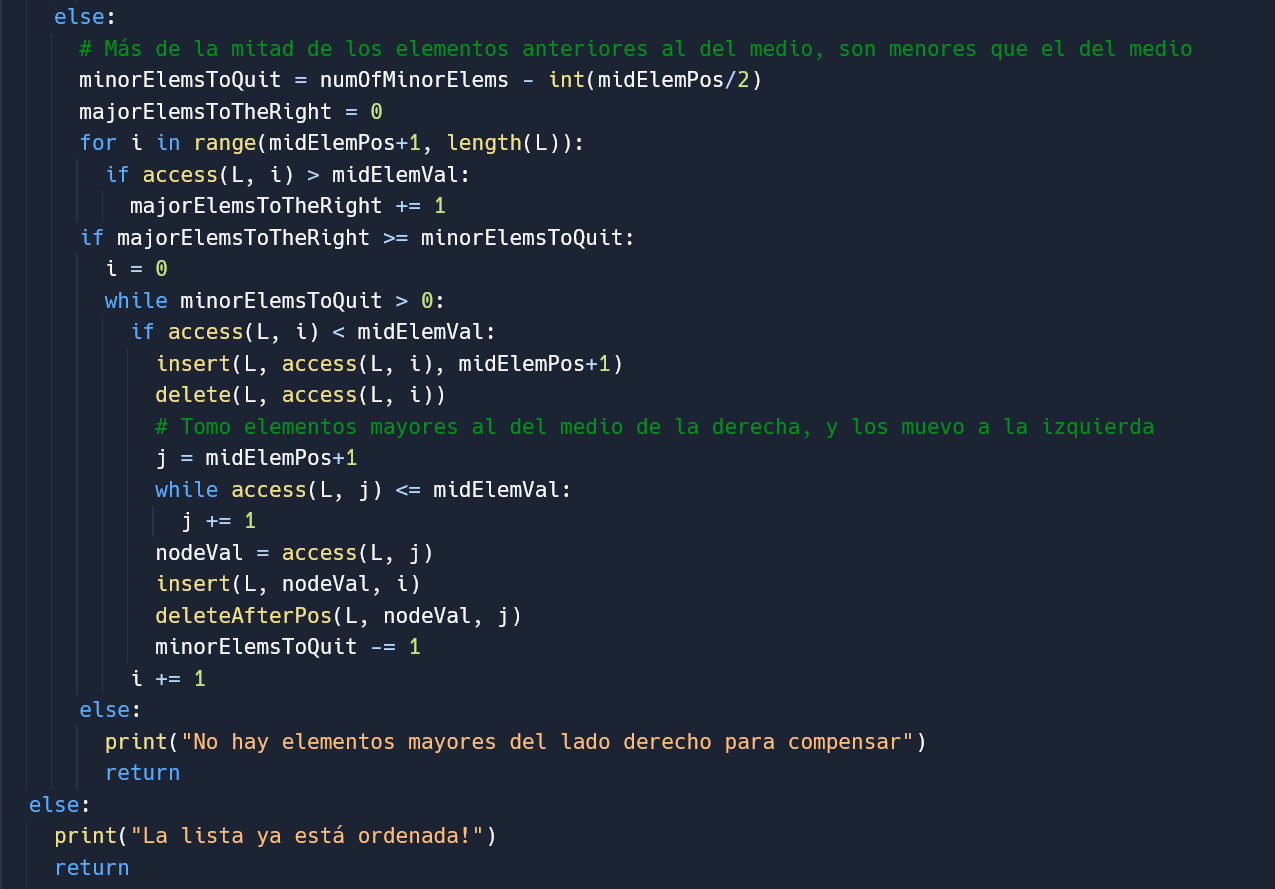
**Ejercicio 4:**

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**

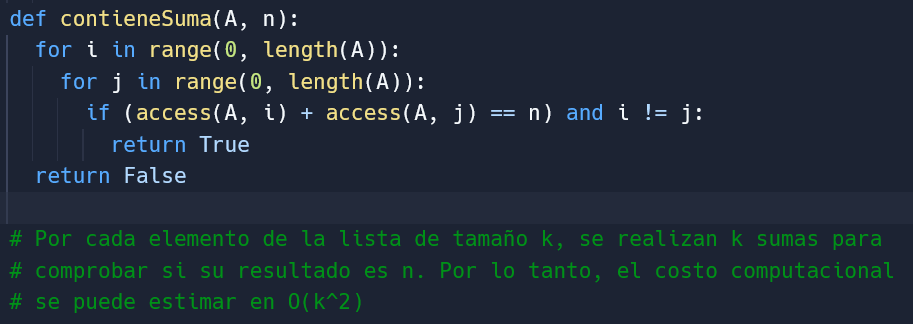
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 3 | 2 | 8 | 5 | 4 | 1 | 6 | 10 | 9 |





Ejercicio 5:

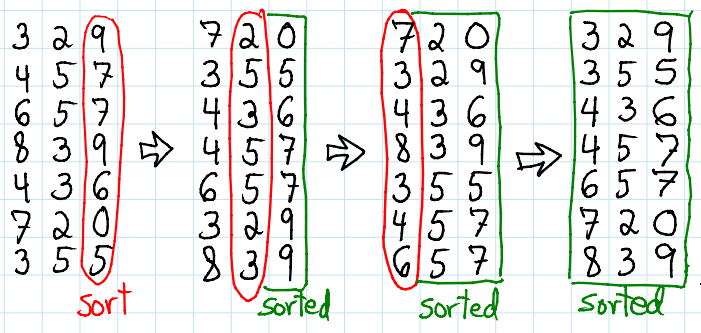
Implementar un algoritmo **Contiene-Suma(A,n)** que recibe una lista de enteros A y un entero n y devuelve True si existen en A un par de elementos que sumados den n. Analice el costo computacional.



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 RadixSort funciona habitualmente ordenando elementos conformados por dígitos, sin compararlos. Para lograr esto, ordena todos los elementos por etapas considerando en primer lugar la cifra menos significativa, ordenando, y repitiendo así cifra por cifra hasta la mayor cifra significativa de todos los elementos.



En la imagen anterior, puede verse ilustrado el funcionamiento del algoritmo. Para un conjunto de 7 números de 3 cifras cada uno, se empieza ordenando la columna ubicada más hacia la derecha, que representa la menor cifra significativa de la base decimal, y se repite el proceso para la columna del medio y la columna izquierda. Para ordenar cada columna, sin embargo, RadixSort se vale de un algoritmo de ordanación intermedio, que debe ser estable. Generalmente, el algoritmo elegido en este caso es CountingSort. El hecho de que sea estable, significa que en cada paso al ordenar, preserva el orden relativo de elementos de igual valor, es decir, el orden en el que aparecen en la lista.

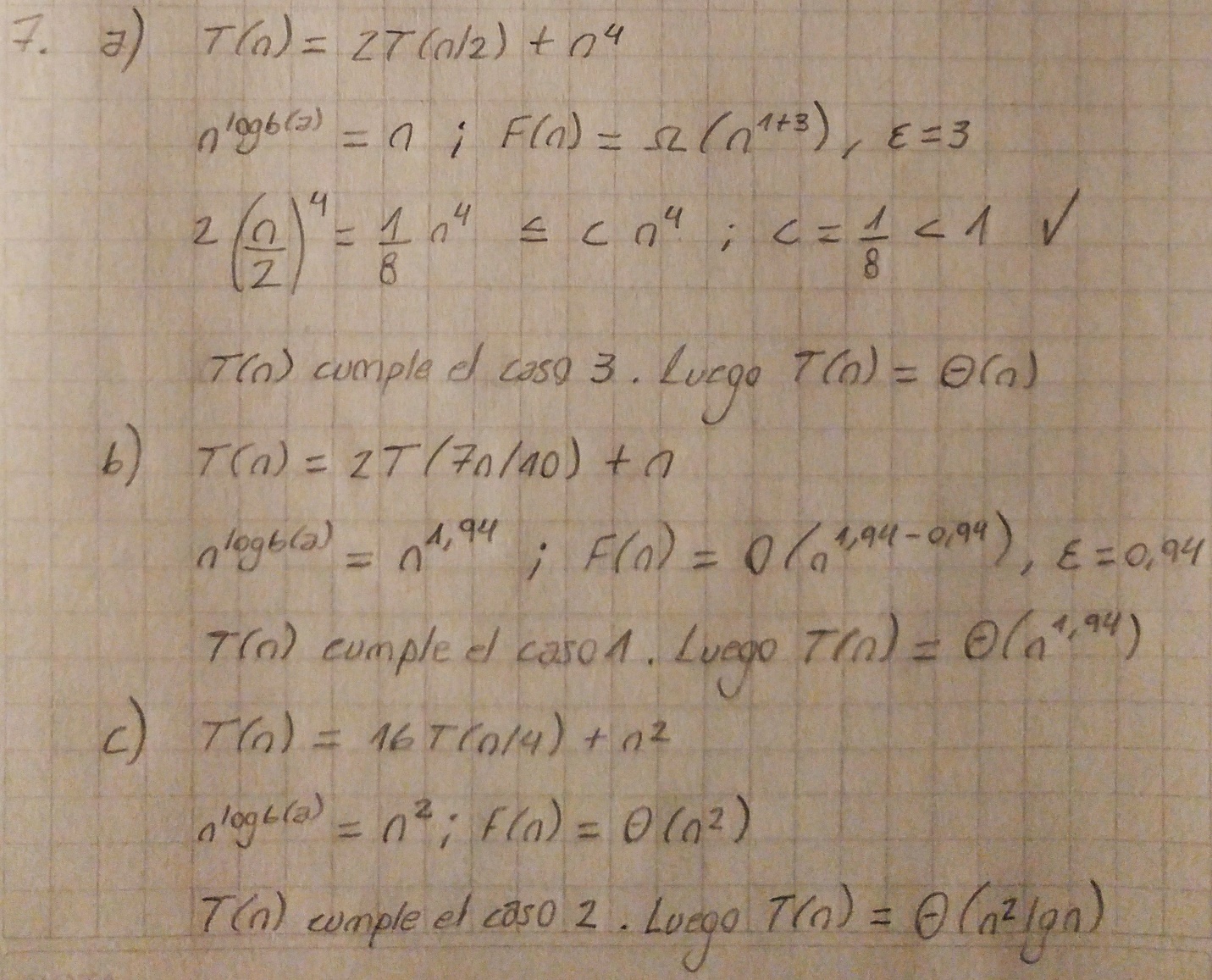
Si los números de la lista a ordenar no tienen todos la misma cantidad de cifras, el algoritmo comienza considerando la cantidad de cifras del mayor elemento de la lista, ya que este determina el número total de pasadas que realizará el algoritmo.

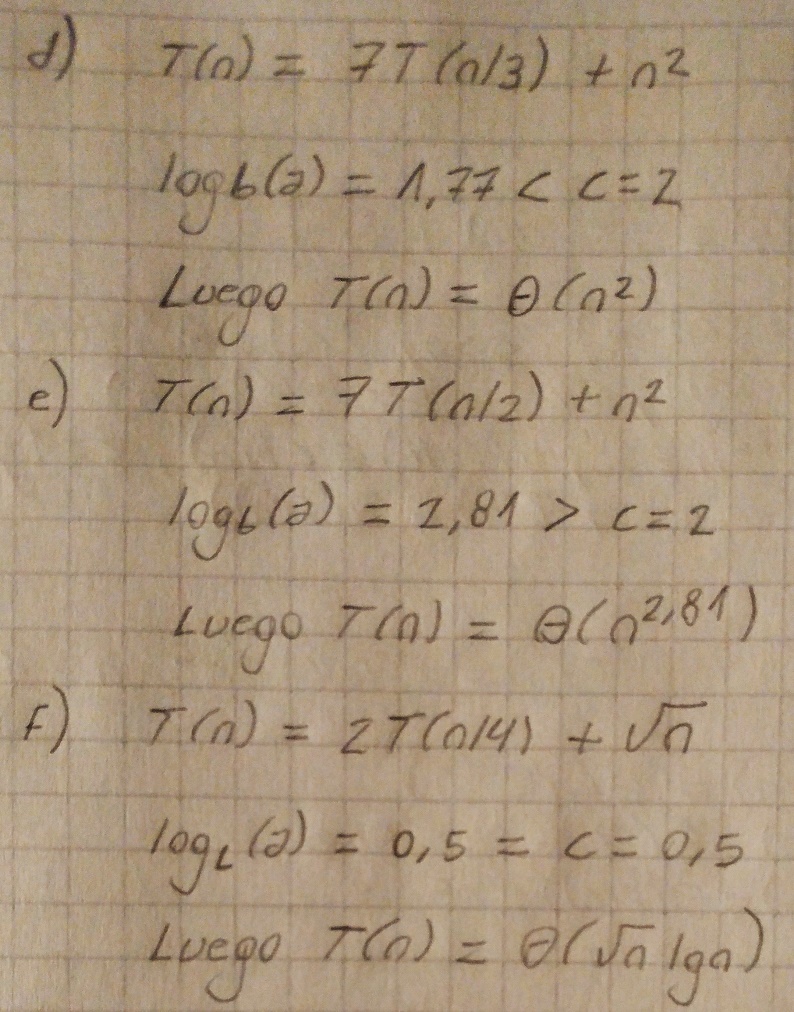
En el caso en que el algoritmo de ordenación intermedio empleado sea CountingSort, la complejidad temporal de RadixSort para los casos mejor, promedio y peor viene dada por O(d\*(n+b)), donde d es el número de dígitos del mayor elemento, y O(n + b) es la complejidad de CountingSort.

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) + nc

1. T(n) = 2T(n/2) + n4
2. T(n) = 2T(7n/10) + n
3. T(n) = 16T(n/4) + n2
4. T(n) = 7T(n/3) + n2
5. T(n) = 7T(n/2) + n2
6. T(n) = 2T(n/4) + √n





#### A tener en cuenta:

1. **Usen lápiz y papel primero**
2. **~~No se puede utilizar otra Biblioteca mas alla de algo1.py y linkedlist.py~~**
3. **Hacer una análisis por cada algoritmo implementado del caso mejor, el caso peor y una perspectiva del caso promedio.**