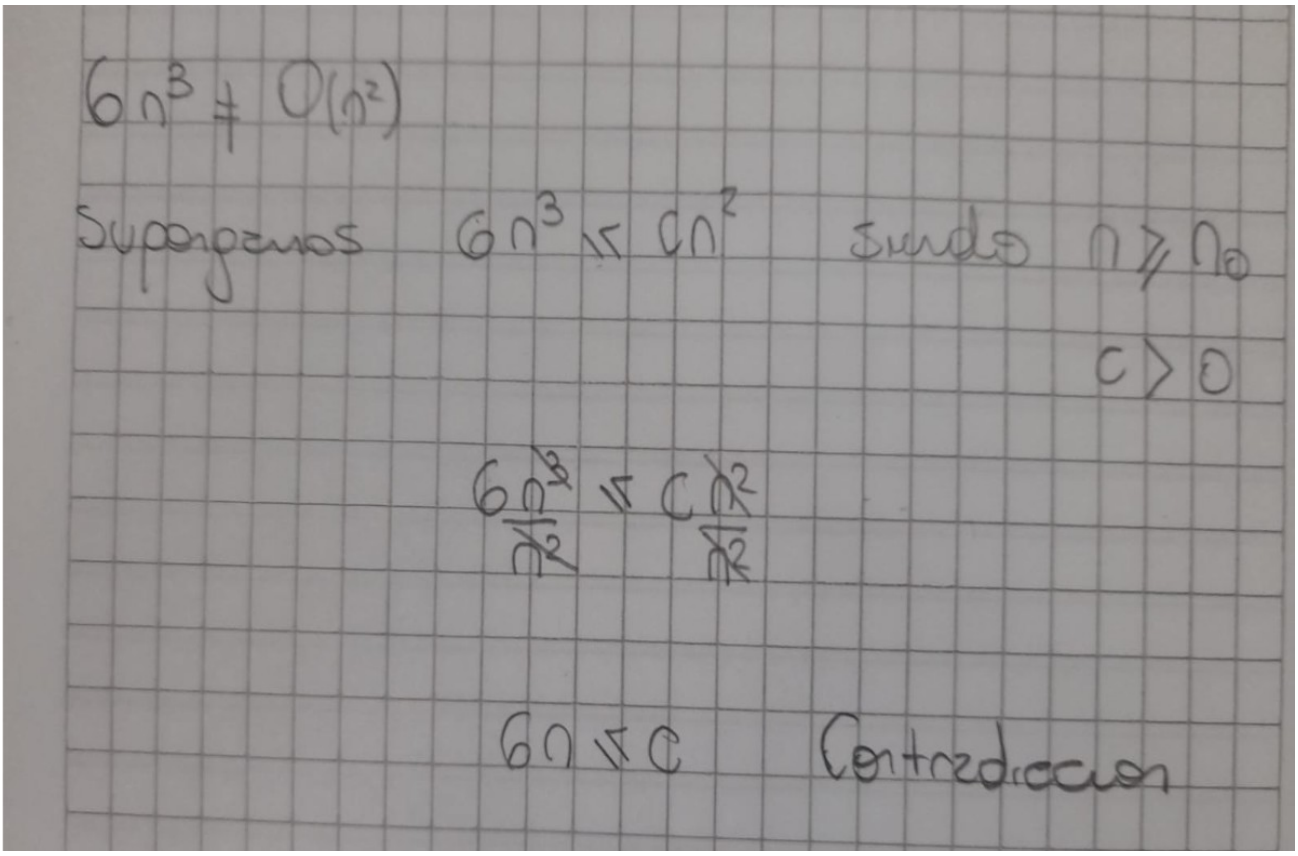
Ejercicio 1:

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



Ejercicio 2:

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

Para el mejor caso, deberiamos elejir un pivote que pueda dividir la lista, de tal manera que quede balanceada. Si vamos a elejir un pivote al azar, que generalmente es el primer elemento.

5	3	2	8	7	4	1	6	10	9

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(n2)

Insertion-Sort(A): O(n)

Merge-Sort(A): O(n log 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 lista de salida

7	3	2	8	5	4	1	6	10	9

```
 5 def exerciseOne(L):
 6     largo = length(L)
 7     currentNode = L.head
 8     menores = LinkedList()
 9     mayores = LinkedList()
10     comparador = access(L, (largo/2) - 1)
11     for i in range(0, largo):
12         if comparador >= currentNode.value:
13             if comparador != currentNode.value:
14                 add(menores, currentNode.value)
15         else:
16             add(mayores, currentNode.value)
17         currentNode = currentNode.nextNode
18         L.head = currentNode
19     addList(L, mayores)
20     add(L, comparador)
21     addList(L, menores)
22
23 def addList(L, m):
24     currentNode = m.head
25     while currentNode != None:
26         add(L, currentNode.value)
27         currentNode = currentNode.nextNode
28
```

Obtengo el nodo de la mitad de la lista, y me traigo un su valor, recorro la lista y compara los valores de la lista con el valor de ese nodo, los que son menores los agrego a la lista de menores y los mayores a la lista de mayores, que previamente eh declarado, y tambien en cada iteracion, vamos eliminando el nodo.

La lista original esta vacia, y le agrego primero la de los menores, y despues la de los mayores y me quedaria la lista los numeros menores que la mitad a la izquierda, lo mayores a la derecha

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.

```
 2
 3 def contieneSuma(A,n):
 4     print(contieneSumaR(A.head, n))
 5
 6 def contieneSumaR(node, n):
 7     if node == None:
 8         return False
 9     aux = node.value
10     nextNode = node.nextNode
11     while node != None:
12         if node.nextNode != None:
13             if (aux + node.nextNode.value) == n:
14                 return True
15         node = node.nextNode
16     return contieneSumaR(nextNode, n)
17
```

$O(n^2)$

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.

BucketSort:

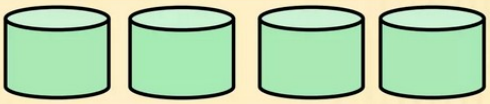
Distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero solo puede contener los elementos que cumplan unas determinadas condiciones. En el ejemplo esas condiciones son intervalos de números. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena individualmente con otro algoritmo de ordenación (que podría ser distinto según el casillero), o se aplica recursivamente este algoritmo para obtener casilleros con menos elementos. Cuando los elementos a ordenar están uniformemente distribuidos el orden de complejidad de este algoritmo es de O(*n*).

Casos:
Peor caso: El peor caso ocurre cuando todos los elementos están en el mismo bucket, lo que significa que los elementos no están distribuidos uniformemente en los diferentes buckets. En este caso, el algoritmo se reduce a O(n^2) debido a que cada bucket se ordena con otro algoritmo de ordenamiento, como por ejemplo, InsertionSort. Entonces, en el peor caso, el tiempo de ejecución es O(n^2).

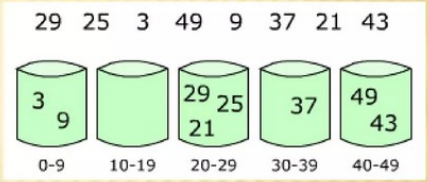
1. Caso promedio: El caso promedio de BucketSort es bastante eficiente y depende en gran medida de cómo estén distribuidos los elementos en los diferentes buckets. Si los elementos se distribuyen uniformemente, entonces el tiempo de ejecución es de O(n + k), donde k es el número de buckets. En la práctica, el número de buckets suele ser pequeño en comparación con n, por lo que el tiempo de ejecución de BucketSort es lineal en la mayoría de los casos.
2. Mejor caso: El mejor caso ocurre cuando los elementos están uniformemente distribuidos en los diferentes buckets y cada bucket tiene el mismo número de elementos. En este caso, cada bucket se ordena con un algoritmo de ordenamiento de tiempo constante, como CountingSort, y el tiempo de ejecución es de O(n). Este es el mejor caso posible para BucketSort y es el más deseado en la práctica.

El algoritmo contiene los siguientes pasos:

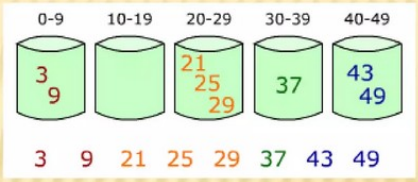
1. *Crear una colección de casilleros vacíos*



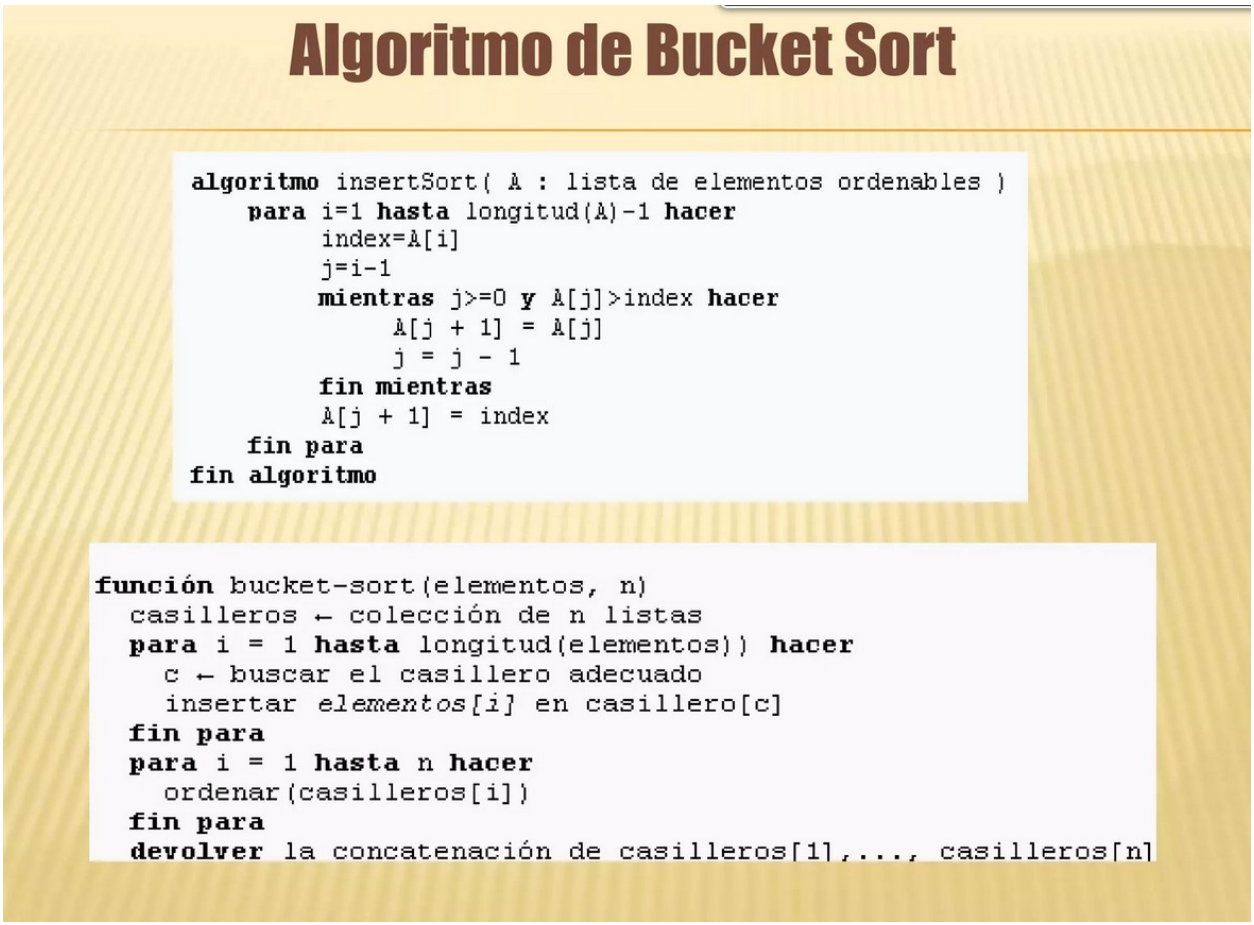
2. *Colocar cada elemento a ordenar en un único casillero*



3. *Ordenar individualmente cada casillero*



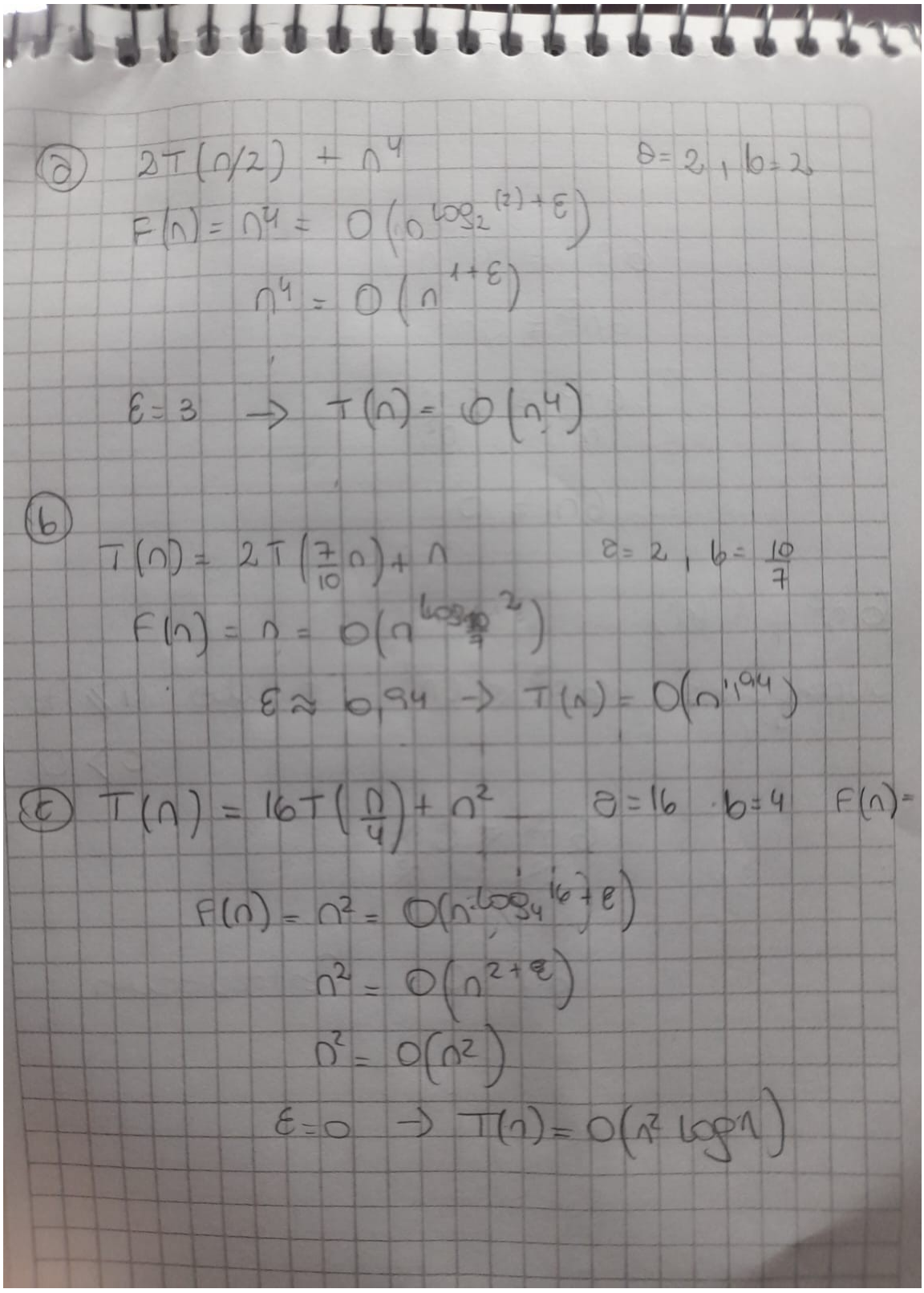
4. *Devolver los elementos de cada casillero concatenados por orden*

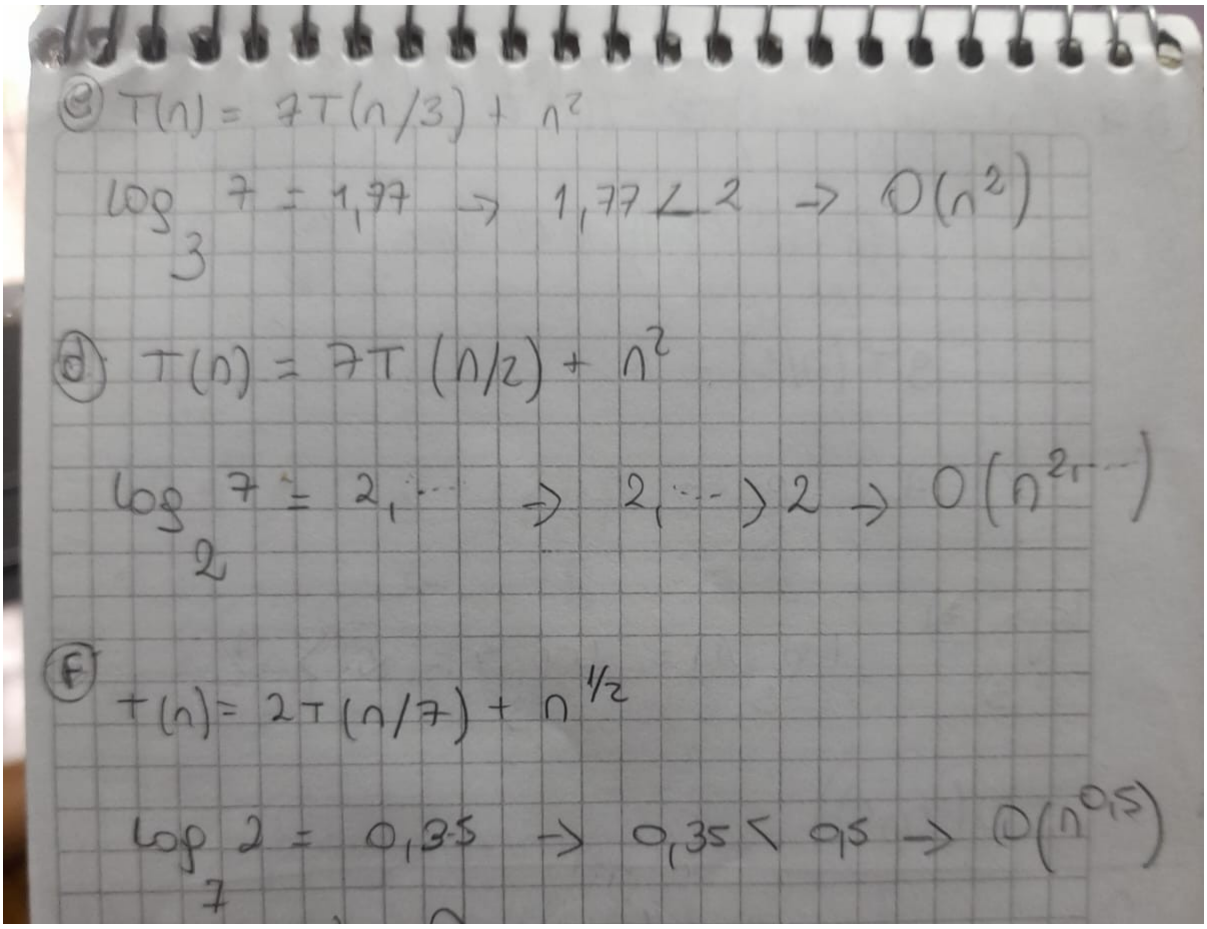


Ejercicio 7:

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) $2T(n/2) + n^4$ $a=2, b=2$
$F(n) = n^4 = O(n^{\log_2(2)+\varepsilon})$
$n^4 = O(n^{1+\varepsilon})$
$\varepsilon = 3 \rightarrow T(n) = O(n^4)$
b) $T(n) = 2T(\frac{7}{10}n) + n$ $a=2, b=\frac{10}{7}$
$F(n) = n = O(n^{\log_{\frac{10}{7}} 2})$
$\varepsilon \approx 0,94 \rightarrow T(n) = O(n^{1,94})$
c) $T(n) = 16T(\frac{n}{4}) + n^2$ $a=16$ $b=4$ $F(n)=$
$F(n) = n^2 = O(n^{\log_4 16 + \varepsilon})$
$n^2 = O(n^{2+\varepsilon})$
$n^2 = O(n^2)$
$\varepsilon = 0 \rightarrow T(n) = O(n^2 \log n)$



Ordenamiento:
1.b

2.f

3.d

4.e

5.c

6.a

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.**