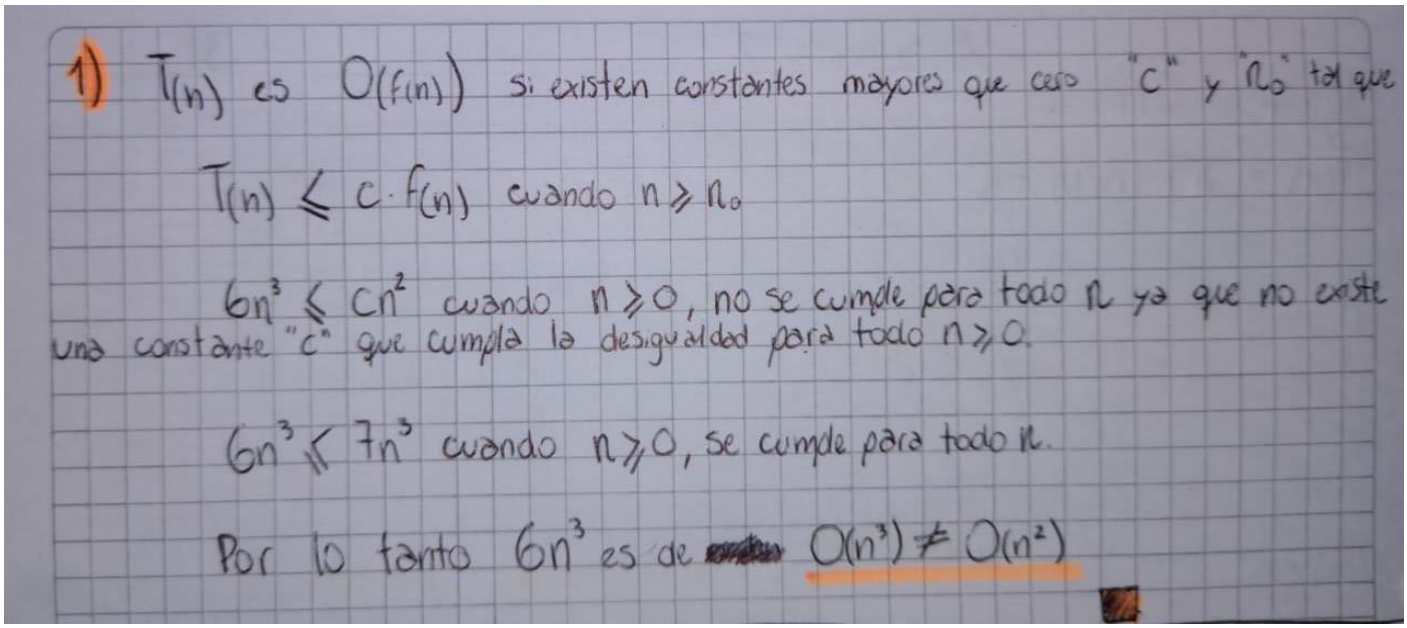
Ejercicio 1:

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) ?

En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es **O(n·log n).**

Array:

6	**2**	**4**	**1**	**5**	**9**	**15**	**10**	**13**	**32**	**11**

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 entrada

7	3	2	8	5	4	1	6	10	9

```
def middle_value_sort(array):
  middle_index = int(len(array)/2)
  print("middle value:", array[middle_index])
  minor_values = 0
  major_values_positions = LinkedList()
  minor_values_positions = LinkedList()

  for i in range(0, middle_index):
    if(array[i] < array[middle_index]):
      minor_values += 1
      add(minor_values_positions, i)
    elif(array[i] > array[middle_index]):
      add(major_values_positions, i)

  # están los valores menores necesarios a la izquierda
  if(minor_values == int(middle_index/2)):
    return array

  # faltan valores menores a la izquierda
  if(minor_values < int(middle_index/2)):
    for i in range(middle_index+1, len(array)):
      if(array[i] < array[middle_index]):
        if(major_values_positions.head == None):
          return array
```

```
        swapPositions(array, i, major_values_positions.head.value)
        deletePosition(major_values_positions, 0)
        minor_values += 1
      if(i == len(array)-1):
        return array

    if(minor_values == int(middle_index/2)):
      return array

  # hay valores menores de más a la izquierda
  if(minor_values > int(middle_index/2)):
    for i in range(middle_index+1, len(array)):
      if(array[i] >= array[middle_index]):
        swapPositions(array, i, minor_values_positions.head.value)
        deletePosition(minor_values_positions, 0)
        minor_values -= 1
      if(minor_values == int(middle_index/2)):
        return array
      if(i == len(array)-1):
        return array


def swapPositions(array, pos1, pos2):
  temp = array[pos1]
  array[pos1] = array[pos2]
  array[pos2] = temp
  #array[pos1], array[pos2] = array[pos2], array[pos1]
  return array

newArray = Array(9, 0)
for i in range(0, len(newArray)):
  newArray[i] = randint(0, 10)
```

https://replit.com/@Giraudo/Ejercitacion-Complejidad-Giraudo-13855#main.py

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. https://replit.com/@Giraudo/Ejercitacion-Complejidad-Giraudo-13855#main.py

```
4
5  def ContieneSuma(A, n):
6      current = A.head
7      aux = A.head
8      if aux.value+current.nextNode.value==n:
9          return true
10     else:
11         while current != None and aux != None: #o(n) peor caso-mejor caso #O(C)
12             if current.nextnode != None:
13                 if aux.value + current.nextnode.value == n:
14                     return True
15                 current = current.nextNode
16             else:
17                 aux = aux.nextNode
18                 current = aux
19         return
```

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.

Counting sort:

El ordenamiento por cuentas (counting sort en inglés) es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Solo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [*mínimo*, *máximo*], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

Se trata de un algoritmo estable cuya complejidad computacional es O(n+k), siendo n el número de elementos a ordenar y k el tamaño del vector auxiliar (*máximo* - *mínimo*).

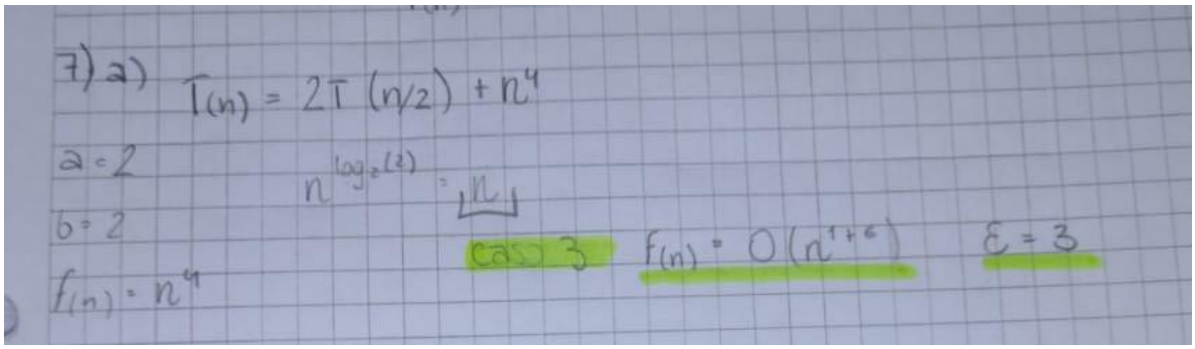
La eficiencia del algoritmo es independiente de lo casi ordenado que estuviera anteriormente. Es decir no existe un mejor y peor caso, todos los casos se tratan iguales.

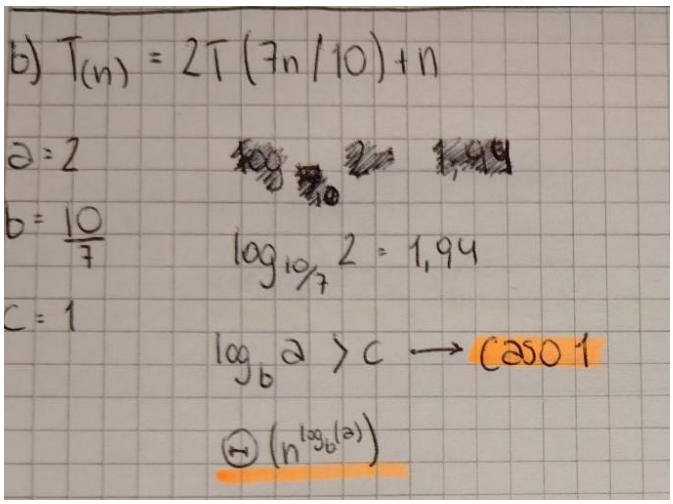
El algoritmo counting, no se ordena **in situ**, sino que requiere de una memoria adicional.

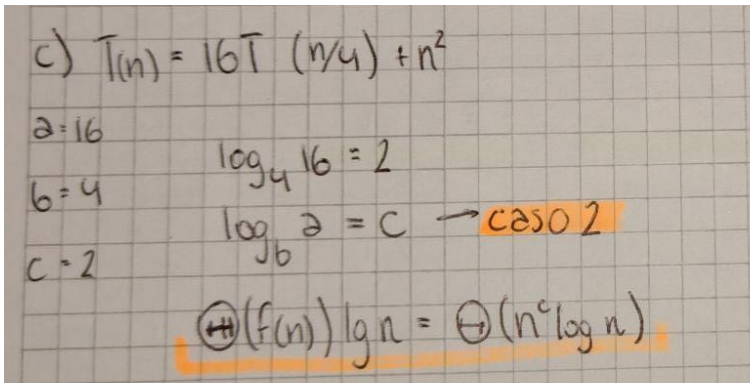
Ejercicio 7:

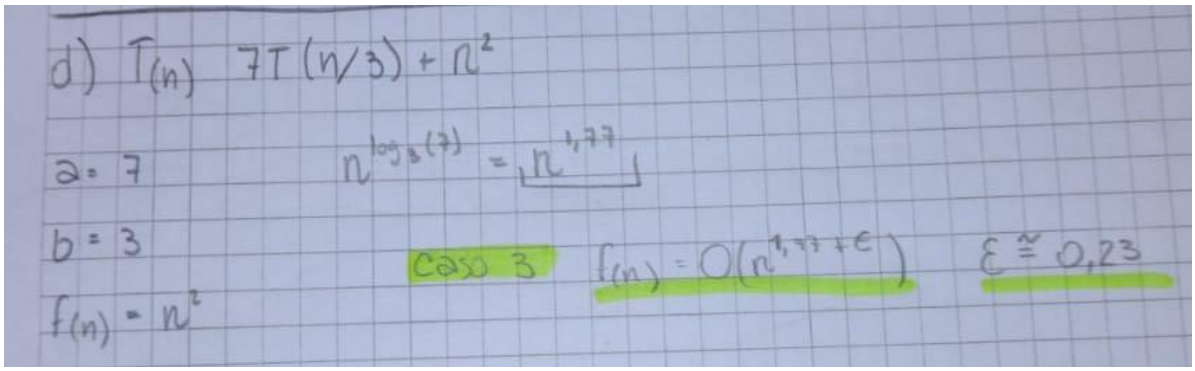
A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: T(n) = a T(n/b) + f(n) y otros 3 con el método maestro simplificado: T(n) = a T(n/b) + n^c

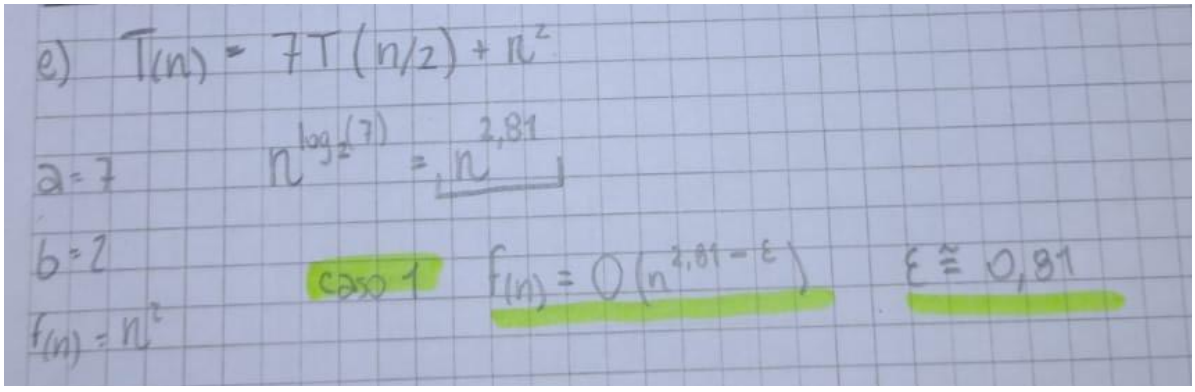
a. $T(n) = 2T(n/2) + n^4$ (6)
b. $T(n) = 2T(7n/10) + n$ (2)
c. $T(n) = 16T(n/4) + n^2$ (5)
d. $T(n) = 7T(n/3) + n^2$ (4)
e. $T(n) = 7T(n/2) + n^2$ (3)
f. $T(n) = 2T(n/4) + \sqrt{n}$ (1)

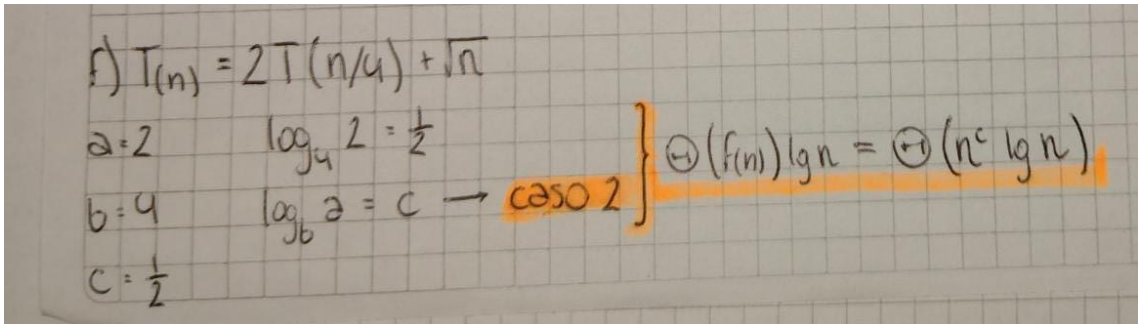





c) T(n) = 16T (n/4) + n²
a = 16
b = 4
c = 2
log4 16 = 2
log_b a = c → caso 2
Θ(f(n)) lg n = Θ(n^c log n)


d) T(n) 7T(n/3) + n²
a = 7
b = 3
f(n) = n²
n^log3(7) = n^1,77
caso 3
f(n) = O(n^1,77+ε)
ε ≅ 0,23


e) T(n) = 7T(n/2) + n²
a = 7
b = 2
f(n) = n²
n^log2(7) = n^2,81
caso 1
f(n) = O(n^2,81-ε)
ε ≅ 0,81


f) T(n) = 2T(n/4) + √n
a = 2
b = 4
c = 1/2
log4 2 = 1/2
log_b a = c → caso 2
Θ(f(n)) lg n = Θ(n^c lg n)