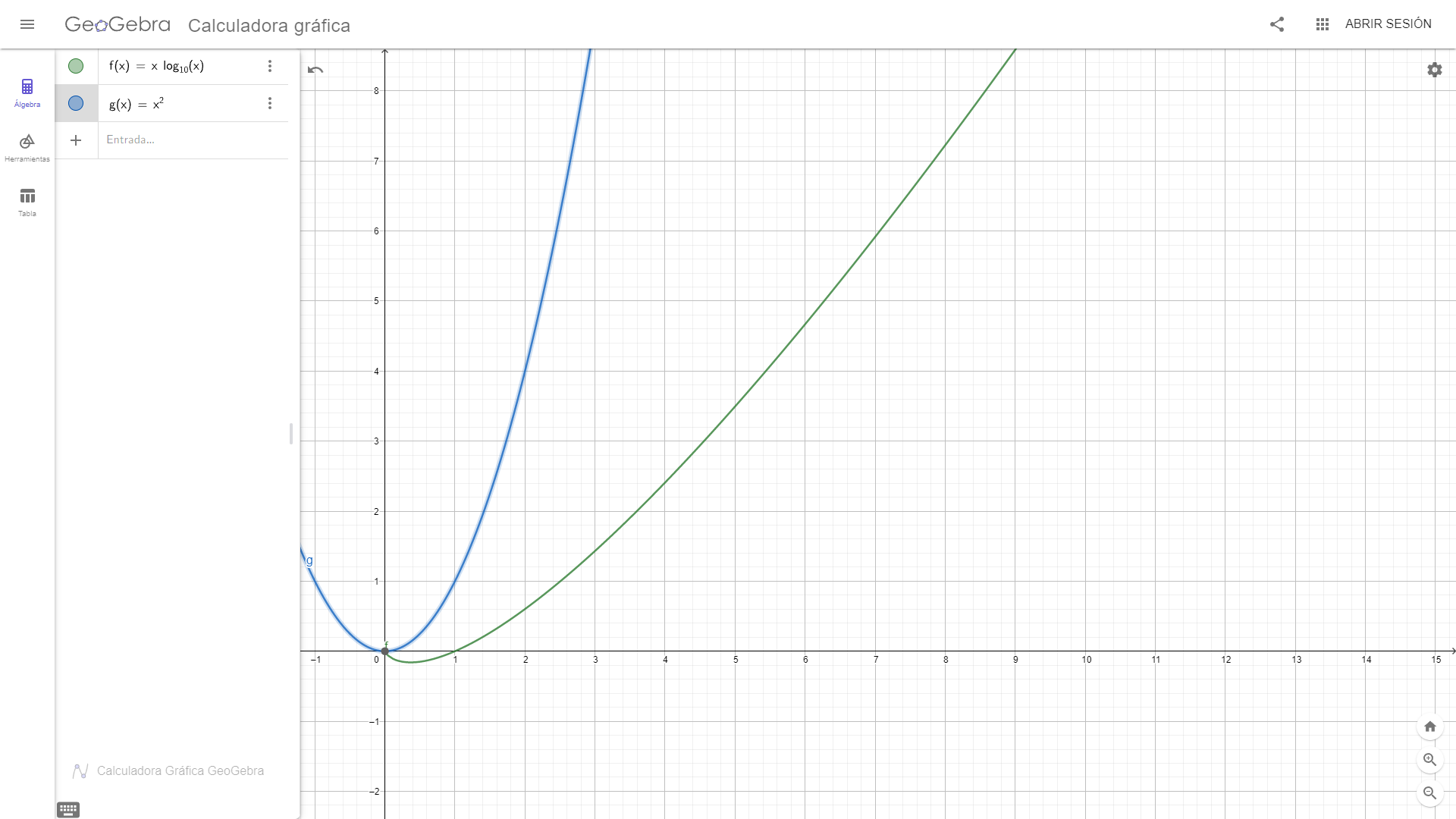
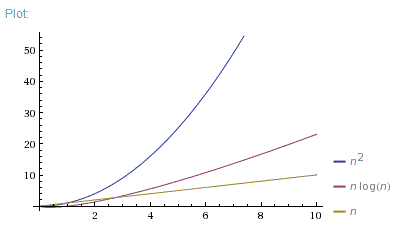
**Deber 1**

**Ejercicio 1**

**complejidad**



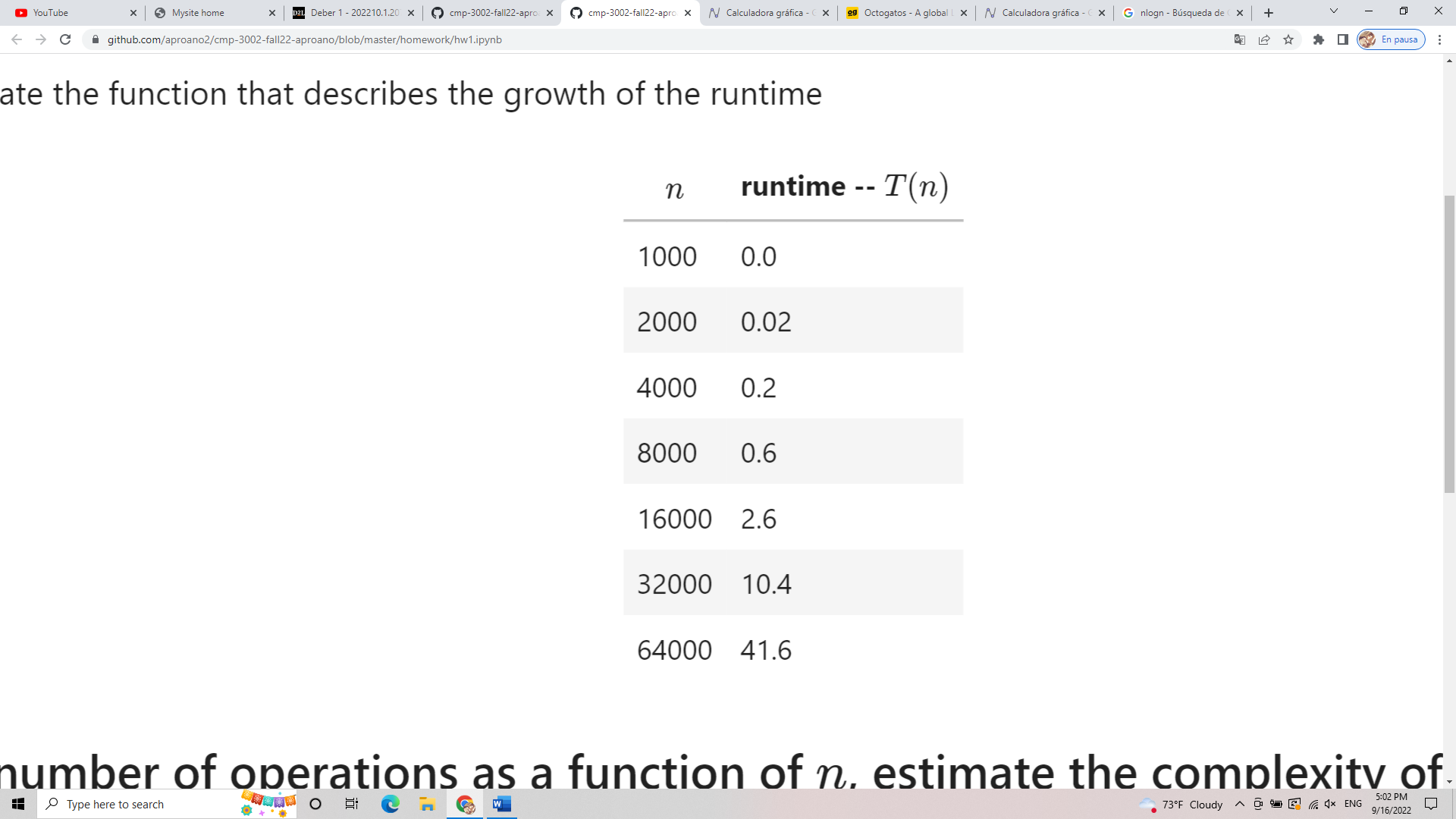
El grafico azul representa una función y el grafico verde una función **logarítmica**, la diferencia entre las 2 funciones es que en la exponencial llegará a cierto punto donde el número de operaciones (eje y) realizadas será más alto que llegar a los resultados (eje x), mientras que un algoritmo que tiene complejidad logarítmica su limite de numero de operaciones es un numero constante porque mientras mas resultados se obtiene, su número de operaciones se estabiliza , es decir para llegar a sacar 8 resultados el computador debió hacer unas 20 operaciones, mientras que para llegar al mismo resultado en una función exponencial, el computador debió hacer un total de aproximadamente 50 operaciones, con esto queda demostrado que un logaritmo es mucho más rápido.



Aquí esta otro grafico donde se compara los distintos tipos de complejidades de mejor manera lo que se describió anteriormente

**Ejercicio 2**

Tiempo de ejecución en base a la tabla



Para este ejercicio se graficaron los resultados en Python usando listas, como se puede apreciar n es el número de operaciones y runtime es el tiempo que se demora en realizar dichas operaciones, para cada movimiento siguiente de la variable **n,** su número se ve duplicado respecto al anterior, por lo que podemos decir que su complejidad es de n\*2, mientras que el tiempo al principio no se mantiene multiplicado por un numero fijo, primero se multiplica por 10, luego por 3 y finalmente se estabiliza en 4 en una cifra mayor que su valor multiplicado por 2, de 0.2 a 0.6 se triplico, de 2.6 a 10.4 se cuadriplico y de 10.4 a 41.6 se quedo en cuadruplicación

0.2 / 0.02 = 10

0.6 / 0.2 = 3

2.6 / 0.6 = 4.333

10.4 / 2.6 = 4

41.6 / 10.4 = 4

import matplotlib.pyplot as plt

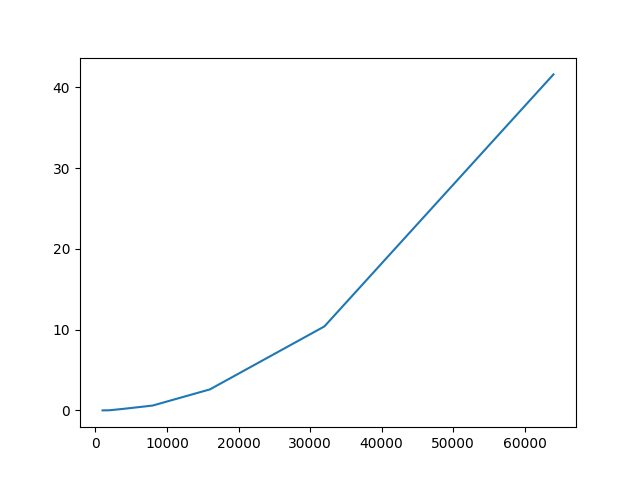
n = [1000,2000,4000,8000,16000,32000,64000,128000]

t = [0.0,0.02,0.2,0.6,2.6,10.4,41.6,83.2]

plt.plot(n,t)

plt.show()

#si crece rapido la funcion al inicio y luego se estabiliza, lo mas probable es que sea una funcion logaritmica



**Ejercicio 3**

Para este ejercicio, las operaciones fuera de los lazos son constantes por lo que no se las toma en cuenta, pero si los for loops y los while ya que repiten **n** veces las operaciones dentro del lazo

El primer for se ejecuta **n** veces

El segundo for esta metido dentro del primero por lo que se multiplica **n** \* **n**

El tercer lazo while esta metido dentro del segundo lazo **for**, sin embargo en el **while** la condición que lo contiene dice mientras que K sea menor a la longitud de la lista A, se van a repetir n veces las operaciones, pero K crece duplicando su valor cada vez que se hace llama la operación de la línea 12, por lo tanto se va a realizar n/2 veces

La operación seria = n^2 \* log n

def func1(n):

    A = range(0,n)

    sum = 0

    i = 0

    for x in A[i:]:  #esto se ejecuta n veces    O(n^2)

        i += 1

        for j in range(i, len(A)):  # esto se ejecuta n-i veces     #loop anidado  for dentro de otro for    O(n)

            y = A[j]

            k = j

            while k < len(A):  #esto se ejecuta (n - 1 - k ) / 2      O(logn)

                z = A[k]

                k = 2\*k

                if x + y <= z:

                    sum += 1

    return sum

#n^2 \* n \* logn

#n^3 \* logn

print(func1(100))

= O (n^3) (logn)

**Ejercicio 4**

from timeit import timeit

import json

import time

inicio = time .perf\_counter()

def sum1(n):

    total = 0

    for i in range(1, n+1):

        total += i

    return total

def sum2(n):

    total = n\*(n+1)//2

    return total

print(sum1(5), " ", sum2(5))

print( sum1(8), " ", sum2(8))

print( sum1(103), " ", sum2(103))

print( sum1(527), " ", sum2(527))

tiempofunc1 = list(range(1,9))

ejeY = list(range(1,9))

e = 1

for x in tiempofunc1:

    print(sum1(e))

    run()

    tiempofunc1[x] = value

    ejeY[x] = e

    e = e\*10 #funcion de n [1,10,100,1000]

tiempofunc2 = list(range(1,9))

ejeY2 = list(range(1,9))

e = 1

for x in tiempofunc2:

    print(sum2(e))

    run()

    tiempofunc2[x] = value

    ejeY2[x] = e

    e = e\*10 #funcion de n [1,10,100,1000]

Aquí se crea las funciones para determinar el tiempo que se demora la función

from functools import wraps

import time

import matplotlib.pyplot as plt

valores = [1,10\*\*1,10\*\*2,10\*\*3,10\*\*4,10\*\*5,10\*\*6,10\*\*7,10\*\*8,10\*\*9]

tiempos1 = []

tiempos2 = []

#decorador de la funcion tiempo para suma1

def timeit1(func):

    @wraps(func)

    def timeit\_wrapper(\*args, \*\*kwargs):

        start\_time = time.perf\_counter()

        result = func(\*args, \*\*kwargs)

        end\_time = time.perf\_counter()

        total\_time = end\_time - start\_time

        tiempos1.append(total\_time)

        print(f'Function {func.\_\_name\_\_}{args} {kwargs} Took {total\_time:.10f} seconds')

        return result

    return timeit\_wrapper

#decorador de la funcion tiempo para suma2

def timeit2(func):

    @wraps(func)

    def timeit\_wrapper(\*args, \*\*kwargs):

        start\_time = time.perf\_counter()

        result = func(\*args, \*\*kwargs)

        end\_time = time.perf\_counter()

        total\_time = end\_time - start\_time

        tiempos2.append(total\_time)

        print(f'Function {func.\_\_name\_\_}{args} {kwargs} Took {total\_time:.10f} seconds')

        return result

    return timeit\_wrapper

@timeit1

def sum1(n):                                                             #costo         veces

    total = 0  # declarar una variable cuesta una operacion              #1             1

    for i in range(1, n+1):  # una variable que se va a llamar N veces   #2             n

        total += i  #sumatoria de loa valores N veces                    #2             n

    return total # declaracion de otra variable de operacion             #1             1

                                                                         #T(n) = 1 + 2n + 2n + 1

                                                                         #T(n) = 4n + 2             #funcion lineal

    #T1(n) = 2c + (c1 + c2)n

    #caso promedio es 10^5

    #T1(n) = 0 + (10^5 + 10^5)n

    # 0 + (10^5 + 10^5)n = 0.0052919000

    # T(n) = 2.64595e-8(n)

    # T(n) = 2.64595e-8(n) \* 10^5

    # T(n) = 0.00264595 segundos

@timeit2

def sum2(n):                                                            #costo          #veces

    total = n\*(n+1)//2  # declaracion de una variable = n \* (n)/2       #4              #1

    return total #declaracion de una variable                           #1              #1

                                                                        #T(n) = 4 + 1      #funcion constante

#T(n) = 5   es una constante

#T(n) = K

#https://www.youtube.com/watch?v=ZD9yICbSyBQ calcular la complejidad de insertion sort

if \_\_name\_\_ == '\_\_main\_\_':

    for i in valores:

        sum1(i)

        sum2(i)

    print(len(tiempos1))

    print(len(tiempos2))

    print(len(valores))

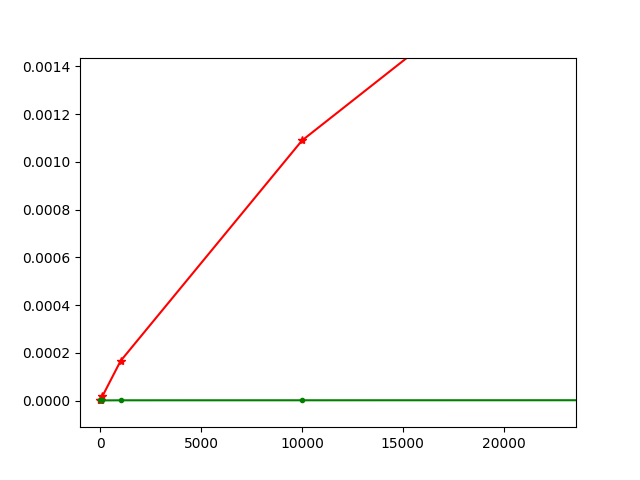
    plt.plot(valores,tiempos1,'r',marker = '\*')  #rojo sumatoria1

    plt.xlim([0, valores[-1]])

    plt.ylim([0, tiempos1[-1]])

    plt.plot(valores,tiempos2,'g',marker = '.')  #verde sumatoria2

    plt.show()



La linea roja es la suma 1 y la línea verde es la suma 2, la función sum1 es menos eficiente y requiere muchas mas operaciones comparado con sum2 que no utiliza for y llega al mismo resultado sin tantas operaciones

Con 10^12, sum1 tardaria dias en llegar a un resultado mientras que sum2

### 5. Prove that the running time of an algorithm is Θ(g(n)) if and only if its worst-case running time is O(g(n)) and its best-case running time is

Θ(g(n)) ----🡪 limite superior = big O = O g(n)

----🡪 limite inferior = Ω (g(n))

Θ(g(n)) = f(n) + f(n) tiene un tiempo de algoritmo promedio = Θ(g(n))

Esto es igual cuando se cumple que c1 \* g(n) ≤ f(n) ≤ c2\*g(n)

Limite superior

Big O : f(n) = O (g(n))

Se cumple cuando:

f(n) ≤ c2 \* g(n)

n -> n≥n0

c2 >0 n0 > 0

ejemplo en 2n+3

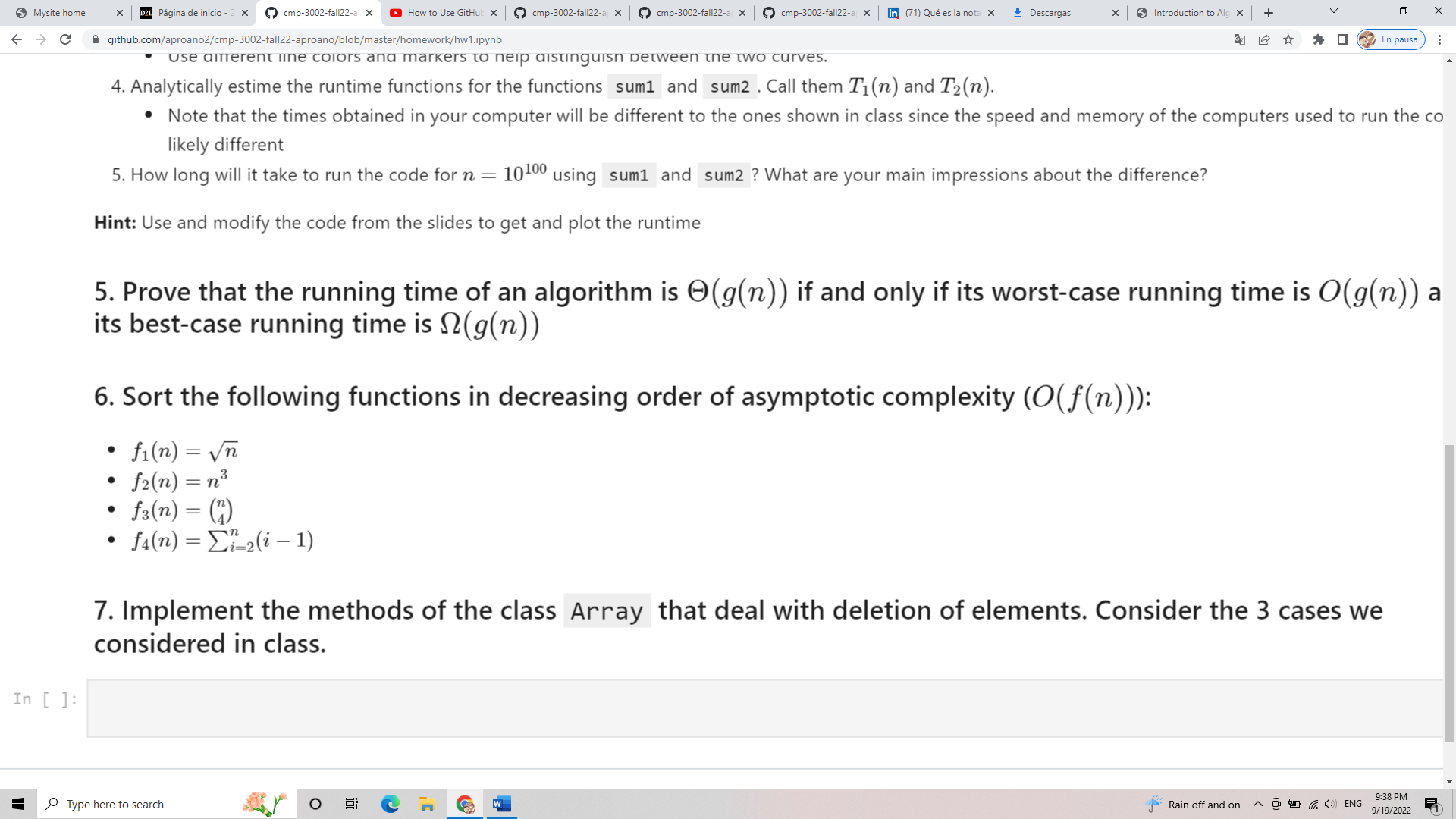
2n+3 ≤ c\*n si c=5 n=x

f(n) ≤ c\*5

límite inferior

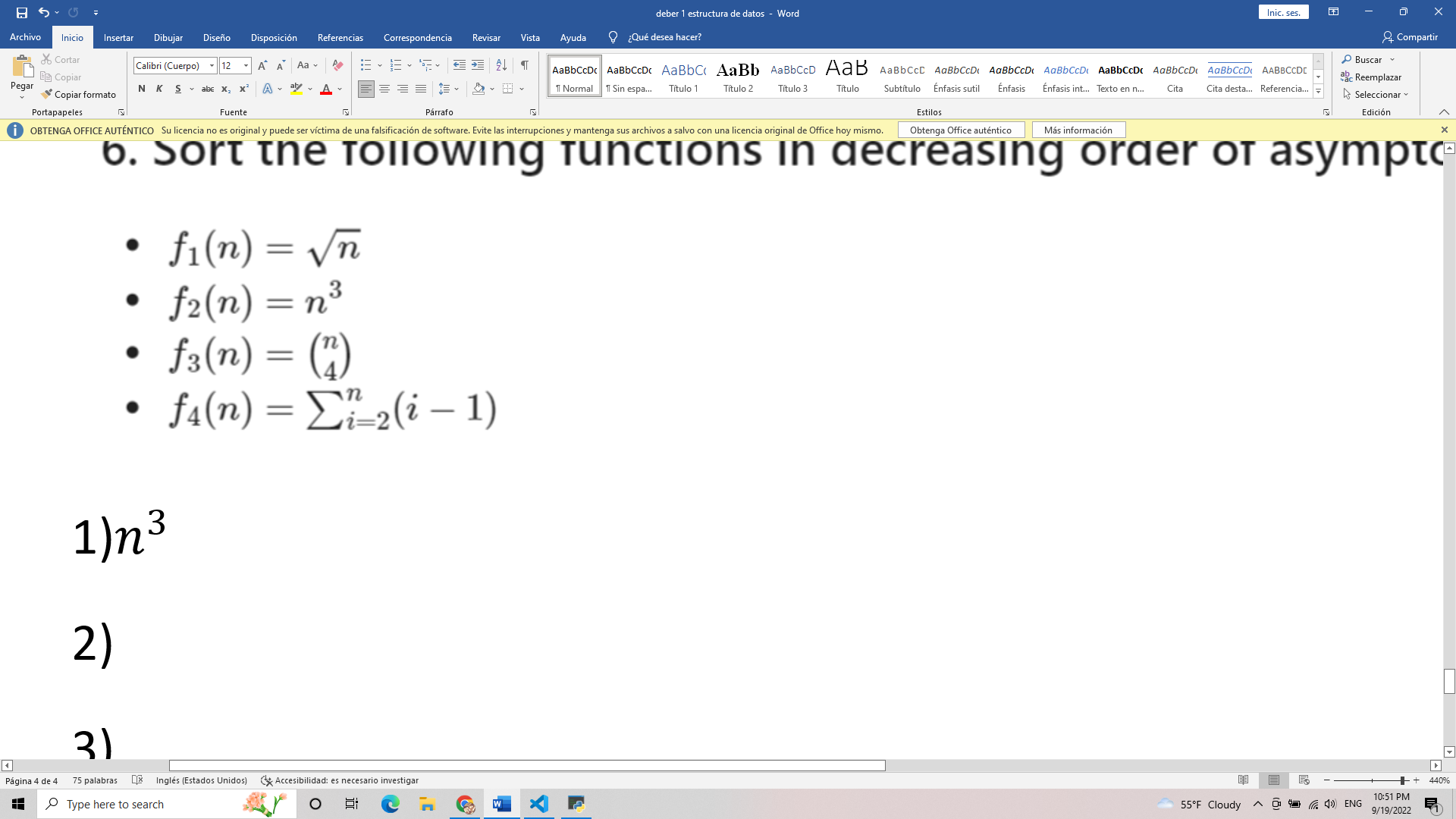
big Ω : f(n) = Ω(g()n)

si se cumple que f(n) ≥ c1 g(n)



1)

2)

3)  = 0(n) funcion lineal

4) numero de posibles combinaciones de 4 elementos en grupo de n elementos

### 7. Implement the methods of the class Array that deal with deletion of elements. Consider the 3 cases we considered in class.

Deleting the last element

Deleting the first element

Delete at any given index

Para este ejercicio se implementan dentro de la clase array utilizando los métodos que se vieron en clase las funciones de remove\_first(), remove\_Index(index), remove\_Last()

Los tres metodos siempre terminan en estas 2 lineas

self.array[self.l-1] = None

        self.l -= 1

representa que la el ultimo espacio del Array que pasa a ser el valor eliminado seleccionado, apunte a NULL porque nuestro método se basa en mover los elementos hacia la izquierda

 ##Remover el primer elemento del array

class Array(Array):

    def remove\_first(self):

    #simplemente para eliminar el primero hay que mover todos los datos hacia la izquierda linea 143

        x = 0

        while x < self.n-1:      # x crece hasta ser igual al ultimo valor dentro de la lista array

            self.array[x] = self.array[x+1]

            x = x+1

   #el ultimo elemento en el array original ya no existe

    #reduce el tamano del array

        self.array[self.l-1] = None

        self.l -= 1

  ##Remover posicion especifica dentro del array

class Array(Array):

    def remove\_Index(self, index):

        # sirve para verificar si el indice ingresado previamente en la funcion esta dentro del rango

        if self.l == self.n:

            raise ValueError("no more capacity")

        if (index < 0) or (index > self.n-1):

            raise IndexError('index out of range!')

        #mueve los datos hacia la izquierda 1 posicion, como en la funcion remove first

        x = index

        while x < self.n-1:

            self.array[x] = self.array[x+1]  # movemos todos los valores una posicion hacia la izquierda comenzando en el indice

            x = x+1

#el ultimo elemento en el array original es null

    #reduce el tamano del array

        self.array[self.l-1] = None

        self.l -= 1

#Remover el ultimo elemento en la lista

class Array(Array):

    def remove\_Last(self):

        self.array[self.l-1] = None

        self.l -= 1

output

1, 2, 3, 4, 5

1, 2, 3, 4, \_ remove last

2, 3, 4, \_, \_ remove first

2, 3, \_, \_, \_ remove index (2)