



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Jesus Cruz Navarro

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 01

No de Práctica(s): 01

Integrante(s): Hernandez Sarabia Jesus Ivan

*No. de Equipo de
cómputo empleado:* No empleado

No. de Lista o Brigada:

Semestre: 2022 - 1

Fecha de entrega: 5 de septiembre de 2021

Observaciones:

CALIFICACIÓN: _____

Objetivo: El estudiante identificara la estructura de los algoritmos de ordenamiento Bubble Sort y Merger Sort

Código fuente:

1) Implementar las funciones que orden una lista de números utilizando los siguientes algoritmos:

a) BubbleSort(arr)

```
bubblesort.py > ...
1  import random # Importamos la biblioteca random para generar numeros aleatorios
2
3  def bubbleSort( arr ): # Funcion bubbleSort para ordenar listas, recibe como argumento una lista
4      n = len(arr)        # contamos el numero de elementos que contiene la lista
5      # Codigo para ordenar los elemetnos de la lista
6      for i in range(n - 1):
7          for j in range(n - 1):
8              if arr[j] > arr[j+1]:
9
10                 aux = arr[j] #Asi se programaria en C
11                 arr[j] = arr[j+1]
12                 arr[j+1] = aux
13
14                 #Sintaxis sugar de python :o
15                 # arr[j], arr[j+1] = arr[j+1], arr[j]
```

b) BubbleSort_Otimizado(arr)

```

1  import random # Importamos la biblioteca random para generar numeros aleatorios
2
3  # Funcion para ordenar los elementos de la lista
4  def bubbleSortOp(arr):
5      n = len(arr)
6      i = 1
7      ordenado = False
8
9      while(i < n and ordenado == False): # Verificamos que la lista contenga elementos para ordenar
10         i = i + 1
11         ordenado = True
12         for j in range( n - 1): # Comenzamos a ordenar la lista que reciba la funcion
13             if(arr[j] > arr[j+1]):
14                 ordenado = False
15                 arr[j], arr[j+1] = arr[j+1], arr[j]
16

```

c) MergeSort(arr, p, r)

```
mersort.py > ...
1  import math # Importamos la biblioteca math para generar calcular math.floor
2  import random # Importamos la biblioteca random para generar numeros aleatorios
3
4  def mergeSort( arr, p, r): #Indice del ultimo elemento r
5      if p < r :
6          q= math.floor((p + r) / 2 )
7          mergeSort( arr, p, q )
8          mergeSort( arr, q+1, r )
9          merge(arr,p,q,r)
10
11 # Funcion merger
12 def merge(arr,p,q,r):
13     # Dividimos la lista a la mitad
14     izq = arr[p: q+1]
15     der = arr[q+1: r+1]
16
17     i = 0
18     j = 0
19
20     # Comenzamos a dividir la lista en elementos mas pequeños para luego comenzar a ordenarlos y juntar
21     # cada elemento en una sola lista ya ordenada
22     for k in range(p, r+1):
23         if (j >= r -q ) or (( i < q-p+1) and izq[i] < der[j]):
24             arr[k] = izq[i]
25             i += 1
26         else:
27             arr[k] = der[j]
28             j += 1
29
```

- 2) Desarrolle un programa que ejecute de cada uno de los algoritmos implementados en el punto 1 con listas de los mismos elementos y obtenga los tiempos de ejecución de cada una de las ejecuciones. Debe imprimir el tiempo de ejecución. No es necesario imprimir en consola las listas.

Bibliotecas que necesitamos para realizar lo solicitado, generar listas aleatorias (random), medir el tiempo que tarda en ordenar cada lista (time), y poder realizar el algoritmo mergesort (math).

```
PracticaP1.py > merge
1  import time
2  import math
3  import random
4
```

Colocamos los algoritmos creados en el punto 1 para poder realizar lo solicitado.

```
PracticaP1.py > merge
5  #Funcion MergeSort
6  def mergeSort( arr, p, r): #Indice del ultimo elemento r
7      if p < r :
8          q= math.floor((p + r) / 2 )
9          mergeSort( arr, p, q )
10         mergeSort( arr, q+1, r )
11         merge(arr,p,q,r)
12
13  # Funcion merger
14  def merge(arr,p,q,r):
15      # Dividimos la lista a la mitad
16      izq = arr[p: q+1]
17      der = arr[q+1: r+1]
18
19      i = 0
20      j = 0
21
22      # Comenzamos a dividir la lista en elementos mas pequeños para luego comenzar a ordenarlos y juntar
23      # cada elemento en una sola lista ya ordenada
24      for k in range(p, r+1):
25          if (j >= r -q ) or (( i < q-p+1) and izq[i] < der[j]):
26              arr[k] = izq[i]
27              i += 1
28          else:
29              arr[k] = der[j]
30              j += 1
31
32  # Funcion bubbleSort
33  def bubbleSort( arr ): # Funcion bubbleSort para ordenar listas, recible como argumento una lista
34      n = len(arr)      # contamos el numero de elementos que contiene la lista
35      # Codigo para ordenar los elemetnos de la lista
36      for i in range(n - 1):
37          for j in range(n - 1):
38              if arr[j] > arr[j+1]:
39
40                  #Sintaxis sugar de python :o
41                  arr[j], arr[j+1] = arr[j+1], arr[j]
```

```
43  # Funcion bubbleSortOptimizada
44  def bubbleSortOp(arr):
45      n = len(arr)
46      i = 1
47      ordenado = False
48
49      while(i < n and ordenado == False): # Verificamos que la lista contenga elementos para ordenar
50          i = i + 1
51          ordenado = True
52          for j in range( n - 1): # Comenzamos a ordenar la lista que reciba la funcion
53              if(arr[j] > arr[j+1]):
54                  ordenado = False
55                  arr[j], arr[j+1] = arr[j+1], arr[j]
```

Creamos una variable que nos indicará cuantos elementos contendrá el arreglo para poder ordenar.

```
PracticaP1.py > ...
56
57     n = 5000 # Numero de elementos que contendra la lista
58     a = [] # Lista
59
```

Para el primer caso que programaremos (caso promedio), necesitaremos el siguiente código para generar una lista aleatoria con números de 0 a 10, con n elementos. Además copiaremos esa misma lista para poder medir cuanto tarda cada algoritmo en ordenar la misma lista.

```
59
60     for i in range(n): # El ciclo se ejecutara tantas veces como elementos tengamos en la lista
61         a.append(random.randint(0, 10)) # Codigo para generar numeros aleatorios del 0 a 10
62
63     a1 = a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
64     a2 = a[:]
65     a3 = a[:]
```

Realizamos el código necesario para poder medir el tiempo que tardara en ordenar cada algoritmo la lista creada.

```
66
67     print("///// Tiempo de ordenamiento para n:", n, " /////\n") # Mostramos cuantos elementos contendra la lista
68     print("----- Caso promedio (Lista aleatoria) -----") # Para caso promedio
69
70
71     print("\n\nTiempo de ordenamiento que tarda MergeSort")
72     inicio = time.time() # Cronometro que nos ayudara a medir cuanto tiempo tarda cada algoritmo en ordenar los elementos de la lista
73     mergeSort(a1, 0, len(a1)-1) # Llamamos al algoritmo para que ordene la lista generada
74     fin = time.time() #Tiempo que tardara el algoritmo en ordenar los elementos de la lista
75     tiempoFinal = fin - inicio # Calculamos el tiempo final restando los cronometros que colocamos anteriormente
76     print(tiempoFinal, " s") # Mostramos el tiempo final
77
78     # Hacemos lo mismo para medir el tiempo de ejecucion de cada algoritmo creado
79
80     print("\n\nTiempo de ordenamiento que tarda BubbleSort")
81     inicio = time.time()
82     bubbleSort(a2)
83     fin = time.time()
84     tiempoFinal = fin - inicio
85     print(tiempoFinal, " s")
86
87     print("\n\nTiempo de ordenamiento que tarda BubbleSort Optimizado ")
88     inicio = time.time()
89     bubbleSortOp(a3)
90     fin = time.time()
91     tiempoFinal = fin - inicio
92     print(tiempoFinal, " s\n\n")
```

Una vez hecho el código para medir cuanto tiempo tarda cada función en ordenar una lista generada aleatoriamente pasamos a programar el mejor caso que es una lista ordenada de manera ascendente, por lo que primero hacemos la lista con elementos ordenados de manera ascendente y copiamos esa misma lista para asignársela a cada algoritmo.

```
PracticaP1.py > ...
91     print(tiempoFinal, " s\n\n")
92
93     print("----- Mejor caso (Lista ordenada asc) -----") # Codigo para medir el mejor caso ordenado asc
94
95     a = []
96
97     for m in range(n): # Creamos la lista ordenada asc para medir los tiempo de ordenamiento
98         a.append(m)
99
100     a1= a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
101     a2= a[:]
102     a3= a[:]
103
```

Realizamos el mismo código anteriormente presentado para medir el tiempo que tardan en “ordenar” la lista

```
103
104 print("\n\nTiempo de ordenamiento que tarda MergeSort")
105 inicio = time.time() # Cronometro que nos ayudara a medir cuanto tiempo tarda cada algoritmo en ordenar los elemetos de la lista
106 mergeSort(a1, 0, len(a1)-1) # Llamamos al algortimo para que ordene la lista generada
107 fin = time.time() #Tiempo que tardara el algoritmo en ordenar los elementos de la lista
108 tiempoFinal = fin - inicio # Calculamos el tiempo final restando los cronometros que colocamos anteriormente
109 print(tiempoFinal, " s") # Mostramo el tiempo final
110
111 # Hacemos lo mismo para medir el tiempo de ejecucion de cada algoritmo creado
112
113 print("\n\nTiempo de ordenamiento que tarda BubbleSort")
114 inicio = time.time()
115 bubbleSort(a2)
116 fin = time.time()
117 tiempoFinal = fin - inicio
118 print(tiempoFinal, " s")
119
120 print("\n\nTiempo de ordenamiento que tarda BubbleSort Optimizado")
121 inicio = time.time()
122 bubbleSortOp(a3)
123 fin = time.time()
124 tiempoFinal = fin - inicio
125 print(tiempoFinal, " s\n\n")
```

Una vez hecho el código para medir cuanto tiempo tarda cada función en ordenar una lista ordenada ascendente pasamos a programar el peor caso que es una lista ordenada de manera descendente, por lo que primero hacemos la lista con elementos ordenamos de manera descendente y copiamos esa misma lista para asignársela a cada algoritmo.

```
127
128 print("----- Peor caso (Lista ordenada desc) -----")
129
130 a = []
131
132 for v in range(n-1, 0, -1): # Creamos la lista ordenada desc para medir los tiempo de ordenamiento
133     a.append(v)
134
135 a1= a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
136 a2= a[:]
137 a3= a[:]
138
```

Nuevamente realizamos el código necesario para poder medir el tiempo que tardara en ordenar cada algoritmo la lista creada.

```
PracticaP1.py > ...
128
129 print("----- Peor caso (Lista ordenada desc) -----")
130
131 a = []
132
133 for v in range(n-1, 0, -1): # Creamos la lista ordenada desc para medir los tiempo de ordenamiento
134     a.append(v)
135
136 a1= a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
137 a2= a[:]
138 a3= a[:]
139
140 print("\n\nTiempo de ordenamiento que tarda MergeSort")
141 inicio = time.time() # Cronometro que nos ayudara a medir cuanto tiempo tarda cada algoritmo en ordenar los elemetos de la lista
142 mergeSort(a1, 0, len(a1)-1) # Llamamos al algoritmo para que ordene la lista generada
143 fin = time.time() #Tiempo que tardara el algoritmo en ordenar los elementos de la lista
144 tiempoFinal = fin - inicio # Calculamos el tiempo final restando los cronometros que colocamos anteriormente
145 print(tiempoFinal, " s") # Mostramo el tiempo final
146
147 # Hacemos lo mismo para medir el tiempo de ejecucion de cada algoritmo creado
148
149 print("\n\nTiempo de ordenamiento que tarda BubbleSort")
150 inicio = time.time()
151 bubbleSort(a2)
152 fin = time.time()
153 tiempoFinal = fin - inicio
154 print(tiempoFinal, " s")
155
156 print("\n\nTiempo de ordenamiento que tarda BubbleSort Optimizado")
157 inicio = time.time()
158 bubbleSortOp(a3)
159 fin = time.time()
160 tiempoFinal = fin - inicio
161 print(tiempoFinal, " s")
```

Una vez finalizado el programa procedemos a ejecutarlo para poder capturar los datos obtenidos, en este caso para $n = 5000$.

```
430
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 1> py PracticaP1.py
///// Tiempo de ordenamiento para n: 5000 /////

----- Caso promedio (Lista aleatoria) -----

Tiempo de ordenamiento que tarda MergeSort
0.06098437309265137 s

Tiempo de ordenamiento que tarda BubbleSort
7.203016757965088 s

Tiempo de ordenamiento que tarda BubbleSort Optimizado
6.897000789642334 s

----- Mejor caso (Lista ordenada asc) -----

Tiempo de ordenamiento que tarda MergeSort
0.055982351303100586 s

Tiempo de ordenamiento que tarda BubbleSort
4.794013738632202 s

Tiempo de ordenamiento que tarda BubbleSort Optimizado
0.0009868144989013672 s

----- Peor caso (Lista ordenada desc) -----

Tiempo de ordenamiento que tarda MergeSort
0.04800868034362793 s

Tiempo de ordenamiento que tarda BubbleSort
9.320014476776123 s

Tiempo de ordenamiento que tarda BubbleSort Optimizado
9.604999780654907 s
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 1> |
```

3.9.6 64-bit 0 0 Live Share

- 3) Pruebe el programa del punto 3 para el mejor caso (lista ordenada asc), peor caso (lista ord desc.) y caso promedio (lista aleatoria) con listas de enteros para $n = 1000, 2000, 5000, 10000$ y 20000 datos. Asegúrese pasar listas con los mismos elementos, es decir, genera una lista y haga copias de esta para ser pasados a los algoritmos (ver ejemplo1). Dado que los tiempos pueden variar, ejecute la prueba 3 veces y obtenga el promedio de los tiempos.

Ejecutamos el programa tres veces por cada valor que debe tomar “n” para así poder obtener el promedio de los tiempos obtenidos de: el caso promedio, mejor caso y peor caso. Tenemos que:

Valores obtenidos para el algoritmo MergerSort:

MergerSort (caso promedio)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.012	0.013	0.017	0.014
2000	0.0238	0.0306	0.0323	0.0289
5000	0.0768	0.0732	0.0693	0.0731
10000	0.1349	0.1163	0.1197	0.12363333
20000	0.2429	0.2429	0.2425	0.24276667

MergerSort (mejor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.011	0.011	0.012	0.01133333
2000	0.04	0.0432	0.0311	0.0381
5000	0.054	0.067	0.062	0.061
10000	0.1012	0.116	0.115	0.11073333
20000	0.2143	0.2143	0.2153	0.21463333

MergerSort (peor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.012	0.01	0.008	0.01
2000	0.024	0.018	0.021	0.021
5000	0.047	0.047	0.0534	0.04913333
10000	0.1061	0.105	0.095	0.10203333
20000	0.1843	0.1844	0.1946	0.18776667

Valores obtenidos para el algoritmo BubbleSort:

Bubblesort (caso promedio)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.314	0.3119	0.315	0.31363333
2000	1.182	1.1429	1.1425	1.1558
5000	8.0714	7.2414	7.0397	7.45083333
10000	29.3023	28.1226	28.3808	28.6019
20000	114.4976	114.4976	116.0483	115.0145

Bubblesort (mejor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.2	0.2297	0.202	0.21056667
2000	0.874	0.7966	0.7659	0.81216667
5000	4.7453	4.865	4.8754	4.82856667
10000	19.8315	20.781	21.9488	20.8537667
20000	76.7511	76.7511	79.4387	77.6469667

Bubblesort (peor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.504	0.4001	0.402	0.43536667
2000	1.59	1.536	1.4511	1.5257
5000	9.5271	11.6104	9.2054	10.1143
10000	43.5304	42.1357	41.6732	42.4464333
20000	181.2549	181.2549	176.0446	179.518133

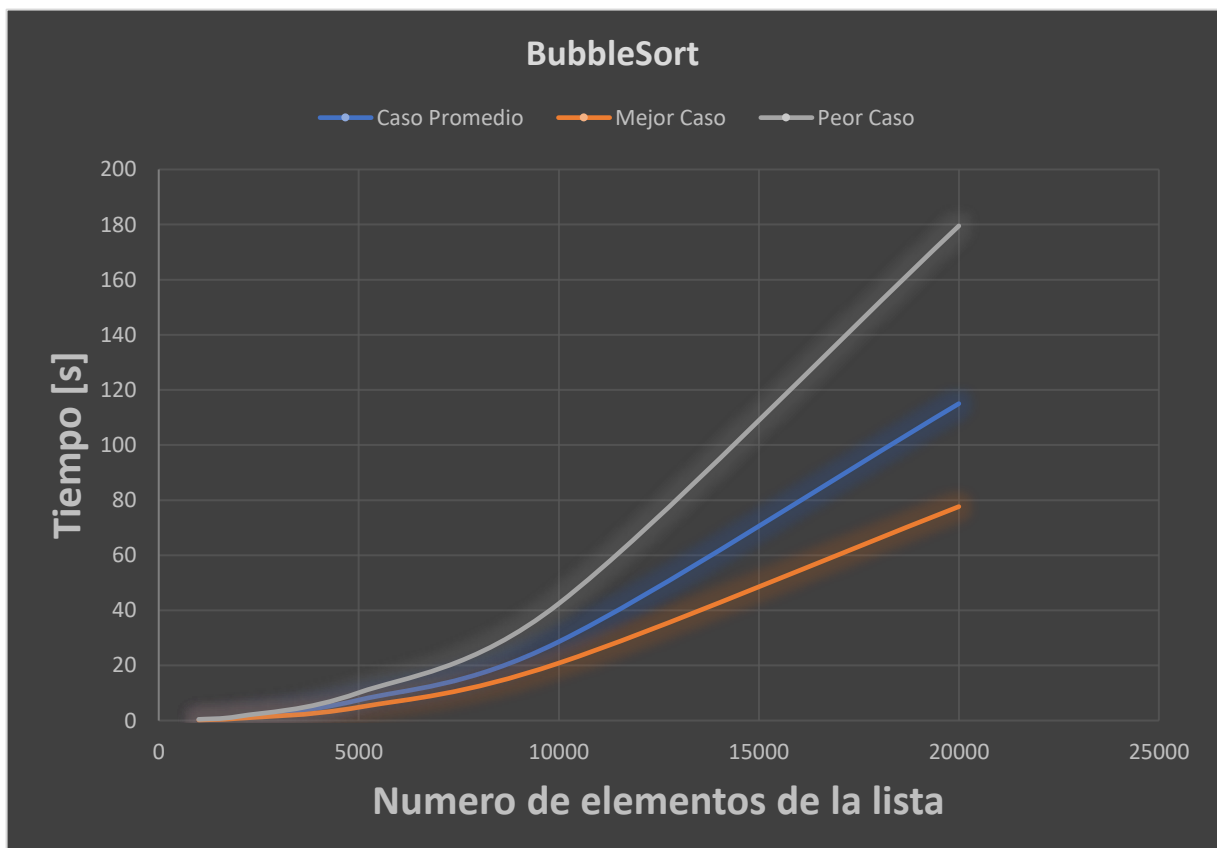
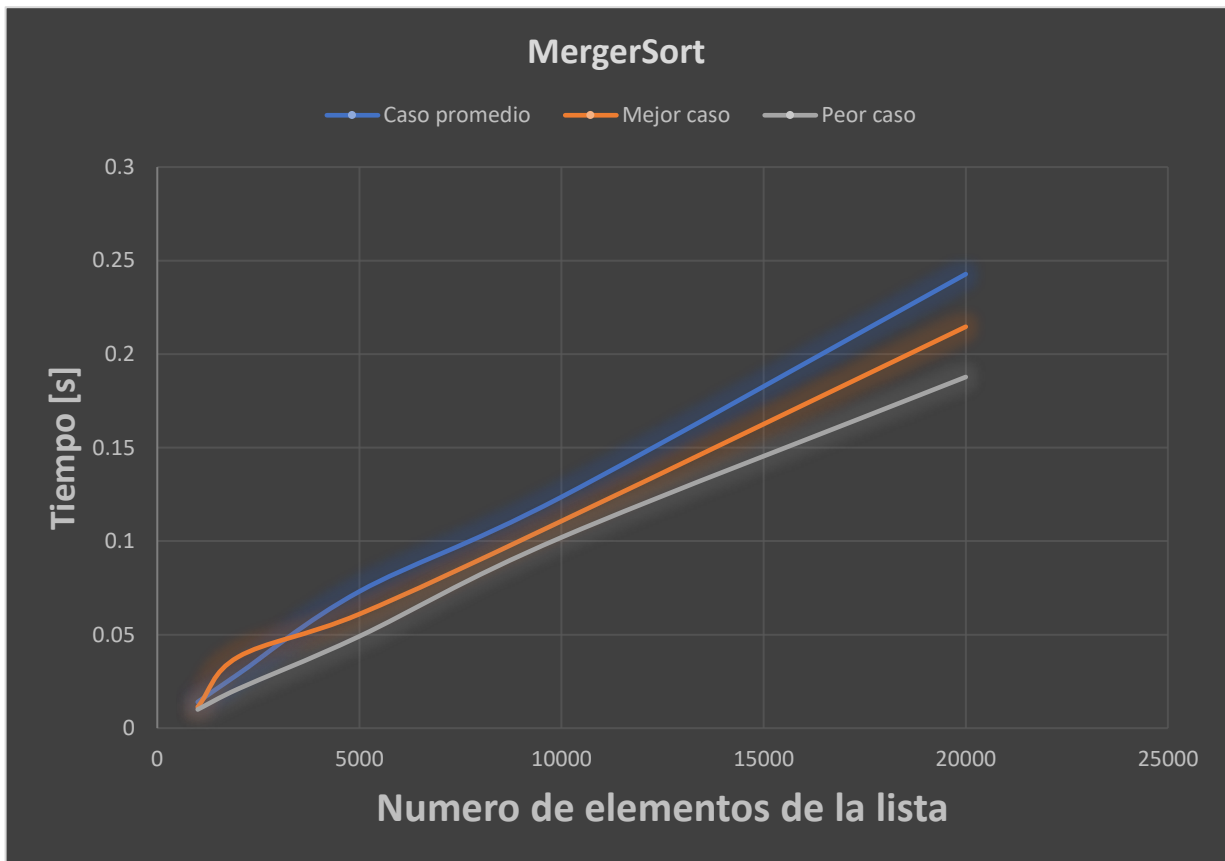
Datos obtenidos para el algoritmo BubbleSort Optimizado:

BubbleSort optimizado (caso promedio)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.293	0.2924	0.292	0.29246667
2000	1.163	1.858	1.0666	1.36253333
5000	6.9076	6.8763	6.7297	6.83786667
10000	26.9247	27.8432	29.4225	28.06346667
20000	109.0054	109.0054	109.2566	109.089133

BubbleSort optimizado (mejor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0	0	0.001	0.00033333
2000	0.001	0	0	0.00033333
5000	0.0156	0	0	0.0052
10000	0.0032	0.0155	0.0079	0.00886667
20000	0.0128	0.0128	0.004	0.00986667

BubbleSort optimizado (peor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.462	0.546	0.603	0.537
2000	1.756	1.6146	1.5676	1.64606667
5000	9.6533	9.6172	9.5015	9.59066667
10000	42.5967	41.5982	46.5774	43.59076667
20000	187.8431	186.9192	187.5153	187.425867

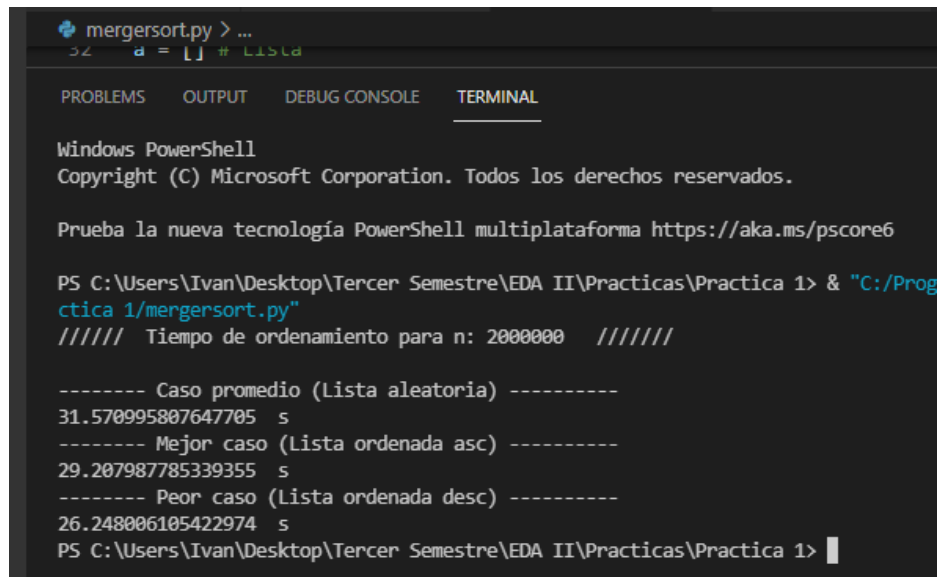
- 4) Grafique, para cada algoritmo los tiempos de ejecución promedio (n vs t) obtenidos en el punto 4 comparando cada uno de los casos. Es decir, deben ser 3 graficas en total, una para cada algoritmo y en cada una deben venir las gráficas de los 3 casos para poder compararlos.





5) Conteste las siguientes preguntas:

a) ¿Cuánto tarda MergeSort en ordenar 2,000,000 de datos?



```
mergesort.py > ...  
a = [] # lista  
  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL  
  
Windows PowerShell  
Copyright (C) Microsoft Corporation. Todos los derechos reservados.  
  
Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6  
  
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 1> & "C:/Prog  
ctica 1/mergesort.py"  
///// Tiempo de ordenamiento para n: 2000000  /////  
  
----- Caso promedio (Lista aleatoria) -----  
31.570995807647705 s  
----- Mejor caso (Lista ordenada asc) -----  
29.207987785339355 s  
----- Peor caso (Lista ordenada desc) -----  
26.248006105422974 s  
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Practicas\Practica 1> |
```

b) ¿Cuánto cree que tardaría BubbleSort en ordenar 2,000,000 de datos?

R: Si tomamos en cuenta que la complejidad del algoritmo se parece mucho a n^2 podemos intentar calcular cuánto se tardaría en ordenar la lista por lo que obtenemos al calcular 2000000^2 obtenemos que tardaría 4×10^{12} segundos en ejecutarse que en minutos es igual a 6.6667×10^{10} que a su vez es igual a 2778 días.

Conclusiones

Hernández Sarabia Jesús Ivan:

Una vez finalizada la practica de laboratorio pudimos comprobar que los algoritmos de divide y vencerás como lo fue MergerSort son mas eficientes a la hora de ordenar listas de numero tanto ordenadas como desordenadas, a comparación de algoritmo de fuerza bruta como lo es BubbleSort.

De igual manera logramos programar una versión optimizada de Bubblesort que si bien no hay mucha diferencia de tiempo en ordenar una lista generada aleatoriamente o ordenada de manera descendente, este verifica mucho más rápido cuando se nos presenta una lista ordenada de manera ascendente.

Y para finalizar pudimos comprobar la complejidad de cada algoritmo implementado, esto se ve reflejado en las graficas trazadas en el punto 4, donde podemos apreciar que los algoritmos BubbleSort y BubbleSort Optimizado son de complejidad n^2 ya que estos pareciera que describen una parábola, mientras que la complejidad de Mergersort se comporta casi de manera lineal (complejidad “n”), es muy probablemente que esto sea ocasionado por la forma en la que trabajan estos algoritmos, que como ya dijimos anteriormente estos funcionan como divide y venceras o como fuerza bruta.