



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): 02

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: 17 de septiembre de 2021

Observaciones:

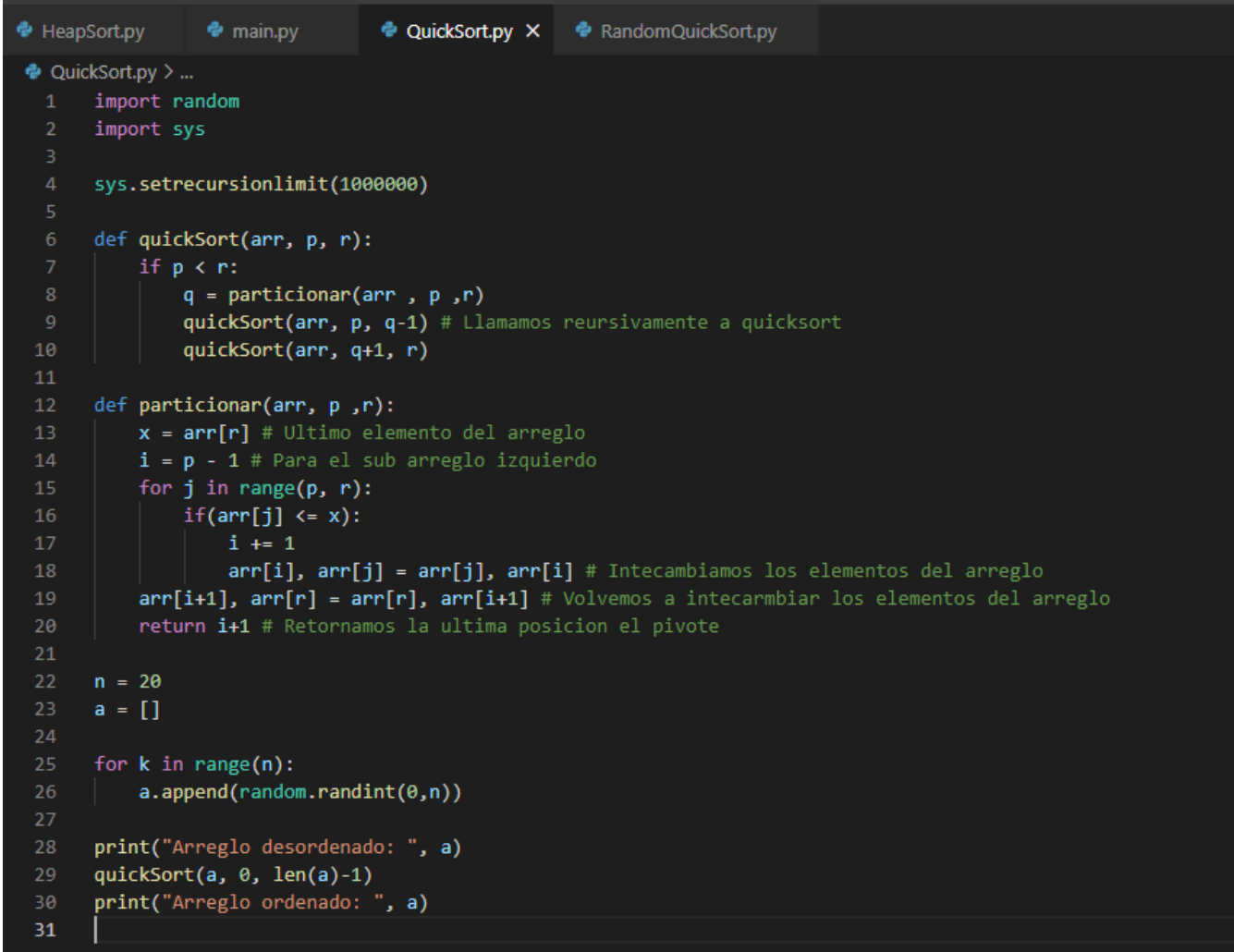
CALIFICACIÓN: _____

Objetivo: El estudiante identificara la estructura de los algoritmos de ordenamiento QuickSort y HeapSort

Código fuente:

1. Implementar las funciones que orden una lista de números utilizando los siguientes métodos:

a. QuickSort(arr. idxIni, idxFin)



```

HeapSort.py  main.py  QuickSort.py X  RandomQuickSort.py
QuickSort.py > ...
1  import random
2  import sys
3
4  sys.setrecursionlimit(1000000)
5
6  def quickSort(arr, p, r):
7      if p < r:
8          q = particionar(arr, p, r)
9          quickSort(arr, p, q-1) # Llamamos reursivamente a quicksort
10         quickSort(arr, q+1, r)
11
12     def particionar(arr, p, r):
13         x = arr[r] # Ultimo elemento del arreglo
14         i = p - 1 # Para el sub arreglo izquierdo
15         for j in range(p, r):
16             if arr[j] <= x:
17                 i += 1
18                 arr[i], arr[j] = arr[j], arr[i] # Intecambiamos los elementos del arreglo
19         arr[i+1], arr[r] = arr[r], arr[i+1] # Volvemos a intecarmbiar los elementos del arreglo
20         return i+1 # Retornamos la ultima posicion el pivote
21
22     n = 20
23     a = []
24
25     for k in range(n):
26         a.append(random.randint(0,n))
27
28     print("Arreglo desordenado: ", a)
29     quickSort(a, 0, len(a)-1)
30     print("Arreglo ordenado: ", a)
31 
```

b. Random_QuickSort(arr, idxIni, idxFin)

```

HeapSort.py  main.py  QuickSort.py  RandomQuickSort.py
RandomQuickSort.py > ...
1  import random
2
3  def rQuickSort(arr, inicio , detener):
4      if(inicio < detener):
5          pivoteIdx = partitionrand(arr, inicio, detener)
6          rQuickSort(arr , inicio , pivoteIdx-1)
7          rQuickSort(arr, pivoteIdx + 1, detener)
8
9  def partitionrand(arr , inicio, detener):
10     randomPivote = random.randrange(inicio, detener)
11
12
13     arr[inicio], arr[randomPivote] = arr[randomPivote], arr[inicio]
14     return partition(arr, inicio, detener)
15
16 def partition(arr,inicio,detener):
17     pivote = inicio
18
19     i = inicio + 1
20
21     for j in range(inicio + 1, detener + 1):
22         if arr[j] <= arr[pivote]:
23             arr[i] , arr[j] = arr[j] , arr[i]
24             i = i + 1
25     arr[pivote] , arr[i - 1] = arr[i - 1] , arr[pivote]
26     pivote = i - 1
27     return (pivote)
28
29 a = []
30 n = 50
31
32 for f in range(n):
33     a.append(random.randint(0, n))
34
35 print("Arreglo desordenado: ", a)
36 rQuickSort(a, 0, len(a)-1)
37 print("Arreglo ordenado: ", a)
38
39
```

c. HeapSort (arr)

```
HeapSort.py X  main.py  QuickSort.py  RandomQuickSort.py
HeapSort.py > ...
1  import random
2  import math
3
4  def MaxHeapify(arr, i, tamañoDelHeap): # funcion maxHepify
5      idxIzq = 2*i # Indice izquierdo del arreglo
6      idxDer = 2*i+1 # Indice derecho del arreglo
7
8      # Comparamos cual es el indice Max
9      if(idxIzq<=(tamañoDelHeap-1) and arr[idxIzq]>arr[i]):
10         posMax=idxIzq
11     else:
12         posMax=i
13     if(idxDer<=(tamañoDelHeap-1) and arr[idxDer]>arr[posMax]):
14         posMax=idxDer
15     # Si posMax es diferente de i intercambiamos los elementos y llamamos recusrivamente a maxHepify
16     if(posMax!=i):
17         intercambia(arr, i, posMax)
18         MaxHeapify(arr, posMax, tamañoDelHeap)
19
20 def construirMaxHeapIni(arr, tamañoDelHeap): # construirMaxHeapIni
21     tamañoDelHeap = len(arr) # tamaño del arreglo
22     for i in range(math.ceil((tamañoDelHeap)/2),-1,-1): # de atras hacia adelante iteramos
23         MaxHeapify(arr, i, tamañoDelHeap) # Llamamos a max heapify para cada uno de los datos
24
25 def heapSort(arr):
26     tamañoDelHeap = len(arr) # Tamaño del arreglo
27     construirMaxHeapIni(arr, tamañoDelHeap) # Construimos el maxHeapIni
28     for i in range(len(arr)-1,0,-1): # iteramos de atras hacia denlante
29         intercambia(arr, 0, i) # Intermabiamos los elementos del arreglo
30         tamañoDelHeap-=1 # Restamos los elemetnos del arreglo que ya estan ordenados
31         MaxHeapify(arr,0,tamañoDelHeap) # Llamamos a maxHepify
32
33 def intercambia(arr, x, y): # Funcion para intecmabiar los elementos del arreglo
34     arr[x],arr[y]=arr[y], arr[x]
35
36 # Main del programa para ordenar los elementos con heapSort
37 a = []
38 elementos = 30
39
40 for k in range(elementos):
41     a.append(random.randint(0,elementos))
42
43 print("Arreglo generado: ", a, "\n")
44 heapSort(a)
45 print("Arreglo ordenado: ", a)
46
```

Ejecución de los programas:

Para QuickSort.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Python + v [ ] [ ] ^ X

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\Laboratorio\Practica 2> & "C:/Program Files/Python39/python.exe" "c:/Users/Ivan/Desktop/Tercer Semestre/EDA II/Laboratorio/Practica 2/QuickSort.py"
Arreglo desordenado: [2, 5, 13, 12, 2, 9, 3, 9, 16, 14, 0, 9, 10, 19, 6, 11, 18, 0, 9, 15]
Arreglo ordenado: [0, 0, 2, 2, 3, 5, 6, 9, 9, 9, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19]
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Laboratorio\Practica 2> |
```

Para Random QuickSort.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Python + v [ ] [ ] ^ X

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\Laboratorio\Practica 2> & "C:/Program Files/Python39/python.exe" "c:/Users/Ivan/Desktop/Tercer Semestre/EDA II/Laboratorio/Practica 2/RandomQuickSort.py"
Arreglo desordenado: [9, 22, 27, 30, 21, 43, 5, 2, 12, 2, 27, 32, 45, 7, 49, 6, 33, 12, 14, 40, 35, 45, 21, 2, 0, 44, 9, 50, 20, 4, 29, 3, 3, 38, 35, 10, 26, 10, 33, 36, 29, 32, 34, 50, 35, 24, 6, 20, 5, 30]
Arreglo ordenado: [0, 2, 2, 2, 3, 3, 4, 5, 5, 6, 6, 7, 9, 9, 10, 10, 12, 12, 14, 20, 20, 21, 21, 22, 24, 26, 27, 27, 29, 29, 30, 30, 32, 32, 33, 33, 34, 35, 35, 35, 36, 38, 40, 43, 44, 45, 45, 49, 50, 50]
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Laboratorio\Practica 2> |
```

Para HeapSort.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Python + v [ ] [ ] [ ] X

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

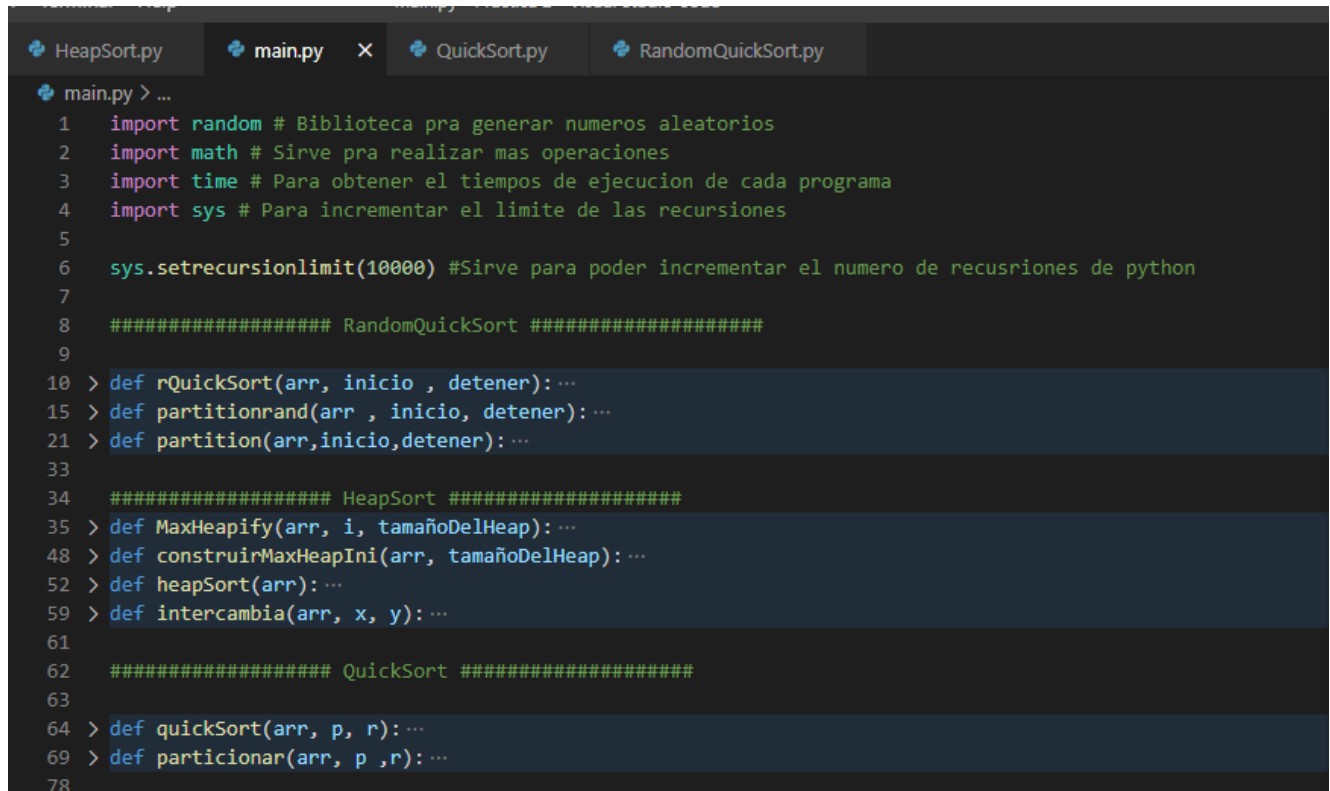
Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Laboratorio\Practica 2> & "C:/Program Files/Python39/python.exe" "c:/Users/Ivan/Desktop/Tercer Semestre/EDA II/Laboratorio/Practica 2/HeapSort.py"
Arreglo generado: [16, 28, 14, 16, 29, 18, 6, 4, 15, 26, 18, 10, 14, 9, 26, 20, 19, 14, 1, 3, 4, 3, 20, 26, 12, 29, 22, 1, 29, 2]

Arreglo ordenado: [1, 1, 2, 3, 3, 4, 4, 6, 9, 10, 12, 14, 14, 14, 15, 16, 16, 18, 18, 19, 20, 20, 22, 26, 26, 26, 28, 29, 29, 29]
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Laboratorio\Practica 2> |
```

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

Primero tendremos que “importar” los algoritmos ya creados en el punto anterior por lo que tendremos lo mostrado en el punto 1, además de las bibliotecas necesarias para poder generar números aleatorios, medir el tiempo de ejecución de cada algoritmo.



```
main.py | main.py | QuickSort.py | RandomQuickSort.py
main.py > ...
1  import random # Biblioteca pra generar numeros aleatorios
2  import math # Sirve pra realizar mas operaciones
3  import time # Para obtener el tiempos de ejecucion de cada programa
4  import sys # Para incrementar el limite de las recursiones
5
6  sys.setrecursionlimit(10000) #Sirve para poder incrementar el numero de recusriones de python
7
8  ##### RandomQuickSort #####
9
10 > def rQuickSort(arr, inicio , detener):...
15 > def partitionrand(arr , inicio, detener):...
21 > def partition(arr,inicio,detener):...
33
34 ##### HeapSort #####
35 > def MaxHeapify(arr, i, tamañoDelHeap):...
48 > def construirMaxHeapIni(arr, tamañoDelHeap):...
52 > def heapSort(arr):...
59 > def intercambia(arr, x, y):...
61
62 ##### QuickSort #####
63
64 > def quickSort(arr, p, r):...
69 > def particionar(arr, p ,r):...
78
```

Una vez realizado lo anterior procederemos a programar los arreglos, para el mejor, peor y caso promedio, y programaremos el código necesario para poder medir el tiempo de ejecución de cada algoritmo para cada caso por lo que obtenemos lo siguiente.

Para el caso promedio tenemos lo siguiente:

```
HeapSort.py  main.py  QuickSort.py  RandomQuickSort.py
main.py > ...
80 ##### Main #####
81 n = 20 # Numero de elementos que contendra la lista
82 a = [] # Lista
83
84 for i in range(n): # El ciclo se ejecutara tantas veces como elementos tengamos en la lista
85     a.append(random.randint(0, n)) #Codigo para generar numeros aleatorios del 0 a n
86
87 a1 = a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
88 a2 = a[:]
89 a3 = a[:]
90
91 print("///// Tiempo de ordenamiento para n:", n, " /////\n") # Mostramos cuantos elementos contendra
92 print("----- Caso promedio (Lista aleatoria) -----") # Para caso promedio
93
94
95 print("\n\nTiempo de ordenamiento que tarda HeapSort")
96 inicio = time.time() # Cronometro que nos ayudara a medir cuanto tiempo tarda cada algoritmo en ordenar l
97 heapSort(a1) # Llamamos al algoritmo para que ordene la lista generada
98 fin = time.time() #Tiempo que tardara el algoritmo en ordenar los elementos de la lista
99 tiempoFinal = fin - inicio # Calculamos el tiempo final restando los cronometros que colocamos anteriorme
100 print(tiempoFinal, " s") # Mostramo el tiempo final
101
102 # Hacemos lo mismo para medir el tiempo de ejecucion de cada algoritmo creado
103
104 print("\n\nTiempo de ordenamiento que tarda QuickSort")
105 inicio = time.time()
106 quickSort(a2, 0, len(a)-1)
107 fin = time.time()
108 tiempoFinal = fin - inicio
109 print(tiempoFinal, " s")
110
111 print("\n\nTiempo de ordenamiento que tarda Random Quick Sort")
112 inicio = time.time()
113 rQuickSort(a3, 0, len(a)-1)
114 fin = time.time()
115 tiempoFinal = fin - inicio
116 print(tiempoFinal, " s\n\n")
117
```

Para el mejor caso:

```
117
118 print("----- Mejor caso (Lista ordenada asc) -----") #Codigo para medir el mejor caso ordenado a
119
120 a = []
121
122 for m in range(n): # Creamos la lista ordenada asc para medir los tiempo de ordenamiento
123     a.append(m)
124
125 a1= a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
126 a2= a[:]
127 a3= a[:]
128
129 print("\n\nTiempo de ordenamiento que tarda HeapSort")
130 inicio = time.time() # Cronometro que nos ayudara a medir cuanto tiempo tarda cada algoritmo en ordenar l
131 heapSort(a1) # Llamamos al algortimo para que ordene la lista generada
132 fin = time.time() #Tiempo que tardara el algoritmo en ordenar los elementos de la lista
133 tiempoFinal = fin - inicio # Calculamos el tiempo final restando los cronometros que colocamos anteriorme
134 print(tiempoFinal, " s") # Mostramo el tiempo final
135
136 # Hacemos lo mismo para medir el tiempo de ejecucion de cada algoritmo creado
137
138 print("\n\nTiempo de ordenamiento que tarda QuickSort")
139 inicio = time.time()
140 quickSort(a2, 0, len(a)-1)
141 fin = time.time()
142 tiempoFinal = fin - inicio
143 print(tiempoFinal, " s")
144
145 print("\n\nTiempo de ordenamiento que tarda Ramdom Quick Sort")
146 inicio = time.time()
147 rQuickSort(a3, 0, len(a)-1)
148 fin = time.time()
149 tiempoFinal = fin - inicio
150 print(tiempoFinal, " s\n\n")
151
```

Para el peor caso:

```
152
153 print("----- Peor caso (Lista ordenada desc) -----")
154
155 a = []
156
157 for v in range(n-1, 0, -1): # Creamos la lista ordenada desc para medir los tiempo de ordenamiento
158     a.append(v)
159
160 a1= a[:] # Copiamos la lista generada para asignarlo a cada algoritmo creado
161 a2= a[:]
162 a3= a[:]
163
164 print("\n\nTiempo de ordenamiento que tarda HeapSort")
165 inicio = time.time() # Cronometro que nos ayudara a medir cuanto tiempo tarda cada algoritmo en ordenar l
166 heapSort(a1) # Llamamos al algortimo para que ordene la lista generada
167 fin = time.time() #Tiempo que tardara el algoritmo en ordenar los elementos de la lista
168 tiempoFinal = fin - inicio # Calculamos el tiempo final restando los cronometros que colocamos anteriorme
169 print(tiempoFinal, " s") # Mostramo el tiempo final
170
171 # Hacemos lo mismo para medir el tiempo de ejecucion de cada algoritmo creado
172
173 print("\n\nTiempo de ordenamiento que tarda QuickSort")
174 inicio = time.time()
175 quickSort(a2, 0, len(a)-1)
176 fin = time.time()
177 tiempoFinal = fin - inicio
178 print(tiempoFinal, " s")
179
180 print("\n\nTiempo de ordenamiento que tarda Random Quick Sort")
181 inicio = time.time()
182 rQuickSort(a3, 0, len(a)-1)
183 fin = time.time()
184 tiempoFinal = fin - inicio
185 print(tiempoFinal, " s")
```


Ejecución del programa:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Python + v [icon] v x
```

```
Copyright (C) Microsoft Corporation. Todos los derechos reservados.
```

Prueba la nueva tecnología PowerShell multiplataforma <https://aka.ms/powershell>

```
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Laboratorio\Practica 2> & "C:/Program Files/Python39/python.exe" "c:/Users/Ivan/Desktop/Tercer Semestre/EDA II/Laboratorio/Practica 2/main.py"
```

```
///// Tiempo de ordenamiento para n: 1000 /////
```

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

```
Tiempo de ordenamiento que tarda HeapSort
0.05400204658508301 s
```

```
Tiempo de ordenamiento que tarda QuickSort
0.020993947982788086 s
```

```
Tiempo de ordenamiento que tarda Random Quick Sort
0.0260007381439209 s
```

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

```
Tiempo de ordenamiento que tarda HeapSort
0.03599977493286133 s
```

```
Tiempo de ordenamiento que tarda QuickSort
0.3319880962371826 s
```

```
Tiempo de ordenamiento que tarda Random Quick Sort
0.01100015640258789 s
```

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

```
Tiempo de ordenamiento que tarda HeapSort
0.03699803352355957 s
```

```
Tiempo de ordenamiento que tarda QuickSort
0.17899656295776367 s
```

```
Tiempo de ordenamiento que tarda Random Quick Sort
0.01007559928894043 s
```

```
PS C:\Users\Ivan\Desktop\Tercer Semestre\EDA II\Laboratorio\Practica 2>
```

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 $n = 1000, 5000, 10000$ y 20000 datos. Dado que los tiempos pueden variar, ejecute la prueba 3 veces y obtenga el promedio de los tiempos.

Datos obtenidos por QuickSort (Tome en cuenta que Quicksort para elementos superiores a 2500 no logra ordenar los elementos de la lista, esto muy probablemente debido al número de recursividades que hace dicho algoritmo para ordenar los elementos)

QuickSort (caso promedio)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
500	0.00999475	0.0019834	0.00299692	0.00499169
1000	0.00399899	0.01599383	0.00999522	0.00999602
1500	0.01001477	0.01598692	0.02099419	0.01566529
2000	0.01600456	0.01800489	0.01900601	0.01767182
2500	0.01699686	0.02801228	0.01799989	0.02100301

QuickSort (mejor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
500	0.26954103	0.10001206	0.08800554	0.15251954
1000	0.27599263	0.27898312	0.24899626	0.26799067
1500	0.53300381	0.52700734	0.53401494	0.53134203
2000	0.95000196	1.13500619	0.92500973	1.00333929
2500	1.51002216	1.43300748	1.70300579	1.54867848

QuickSort (peor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
500	0.04900646	0.0680089	0.06599879	0.06100472
1000	0.19701266	0.19499254	0.24899626	0.21366715
1500	0.35901117	0.3500154	0.37501597	0.36134752
2000	0.59299493	0.61000514	0.60299373	0.60199793
2500	1.10302281	0.92003369	0.96199822	0.99501824

Para random QuickSort

Random QuickSort (caso promedio)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.02300286	0.01899886	0.01299214	0.01833129
2000	0.07600451	0.02700305	0.02200198	0.04166985
5000	0.07201672	0.09398866	0.07500148	0.08033562
10000	0.11400151	0.12701797	0.1430068	0.12800876
20000	0.23501277	0.26000071	0.2458725	0.24696199

Random QuickSort (mejor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.02099109	0.01899934	0.01701117	0.01900053
2000	0.02601075	0.02399302	0.03199887	0.02733421
5000	0.06498575	0.09901261	0.05300832	0.07233556
10000	0.09400034	0.10099769	0.10801244	0.10100349
20000	0.19501543	0.22400045	0.23598719	0.21833436

Random QuickSort (peor caso)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.01098919	0.01998973	0.01100039	0.0139931
2000	0.02200818	0.027004	0.01599526	0.02166915
5000	0.05999446	0.13699532	0.05799007	0.08499328
10000	0.10999584	0.11300087	0.09899306	0.10732992
20000	0.21798396	0.23599863	0.27696586	0.24364948

Para HeapSort

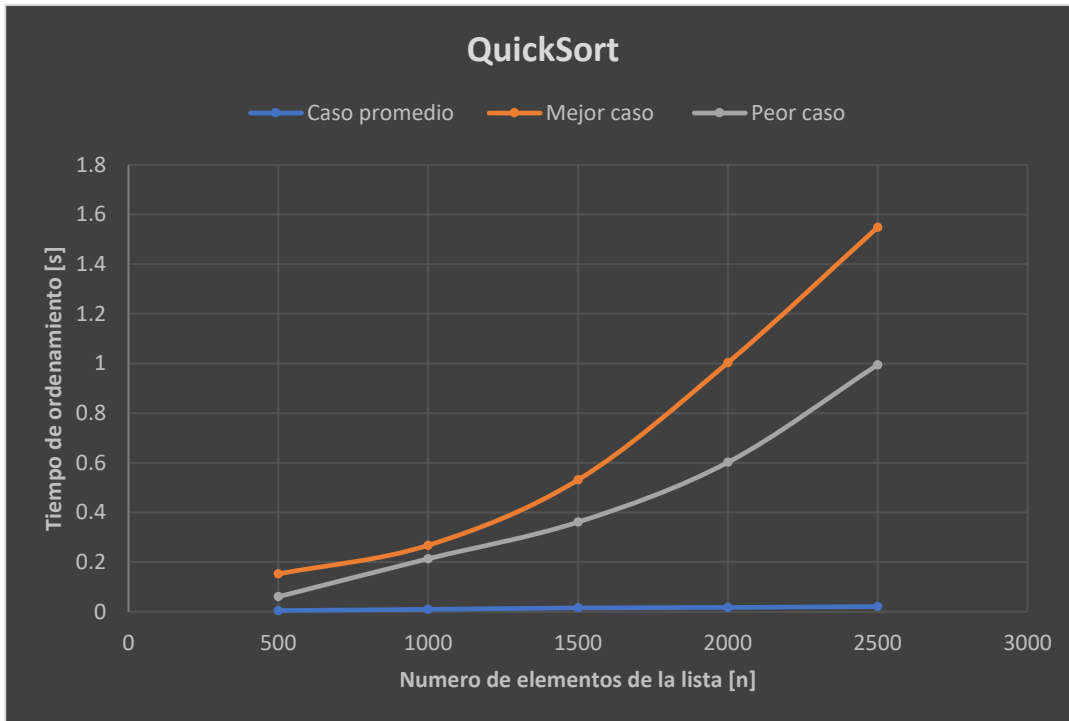
HeapSort (caso promedio)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.01900387	0.03100705	0.02000666	0.02333919
2000	0.05501509	0.06200123	0.07600379	0.06434004
5000	0.13098836	0.14499879	0.12499714	0.13366143
10000	0.26110554	0.24599624	0.24401069	0.25037082
20000	0.51501226	0.58300638	0.57399178	0.55733681

HeapSort (caso mejor)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.04499817	0.0360024	0.0269959	0.03599882
2000	0.07600451	0.06600666	0.10399556	0.08200224
5000	0.11100483	0.13899374	0.12900686	0.12633514
10000	0.24298716	0.24499798	0.24499249	0.24432588
20000	0.51100898	0.57698226	0.57216573	0.55338566

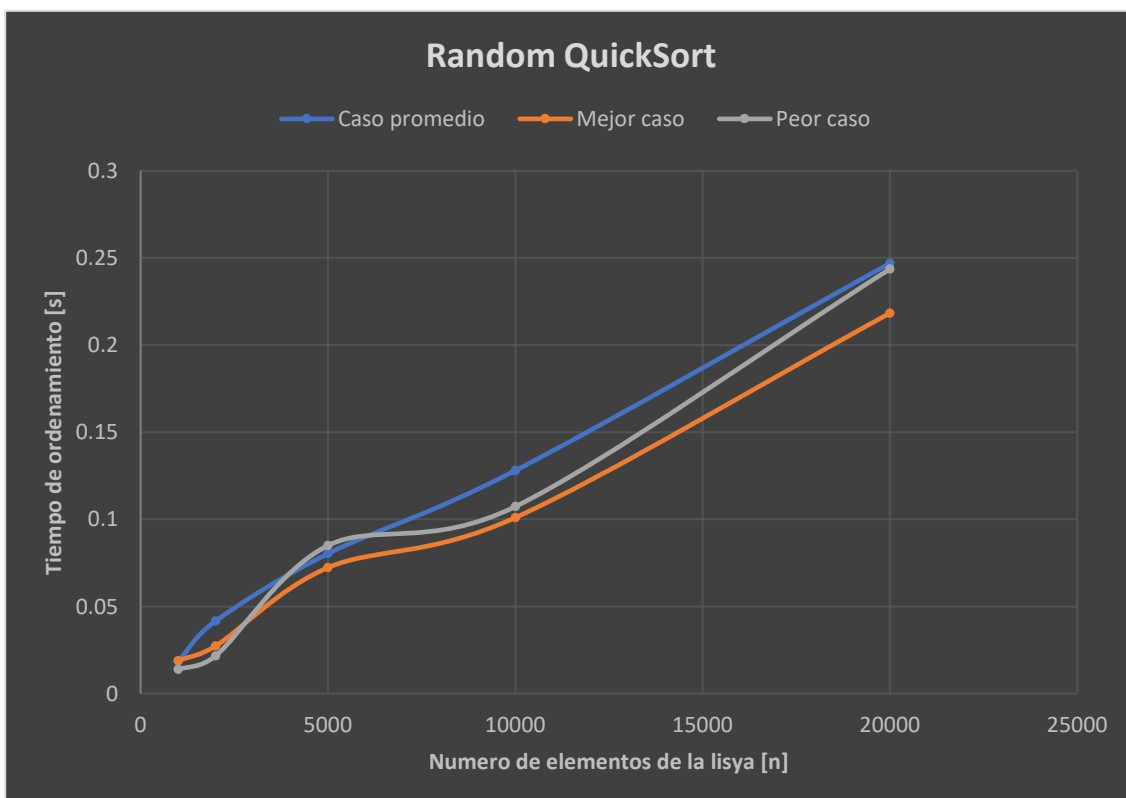
HeapSort (caso peor)				
Para n	Primer valor	Segundo valor	Tercer valor	Promedio
1000	0.03599834	0.0299902	0.02999473	0.03199442
2000	0.06001163	0.05099225	0.05198574	0.05432987
5000	0.12300205	0.13699532	0.11301136	0.12433624
10000	0.21999216	0.22400546	0.2160008	0.21999947
20000	0.46100593	0.54198861	0.54600644	0.51633366

4.- Grafique, para cada caso, los tiempos de ejecución n vs t obtenidos en el punto 3 comparando los 3 algoritmos en la misma gráfica. Es decir, deben ser 3 graficas.

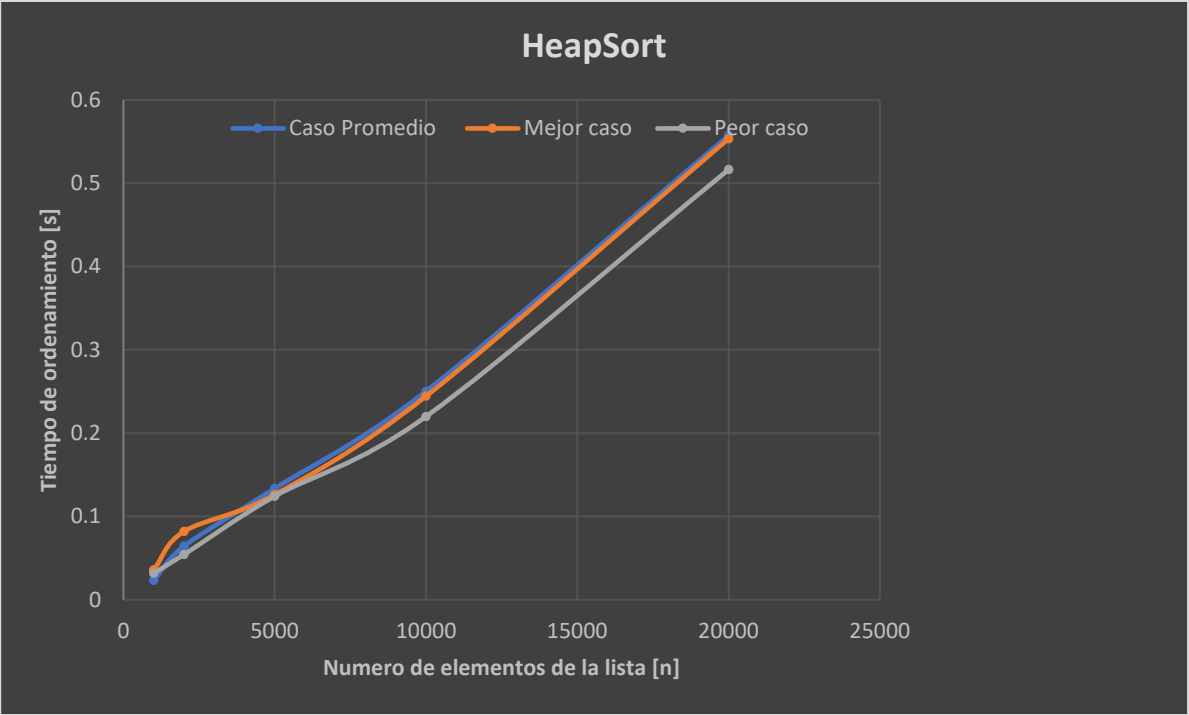
Grafica obtenida para QuickSort



Grafica obtenida para Random QuickSort



Grafica obtenida para HeapSort



Conclusiones:

Hernández Sarabia Jesús Ivan:

Una vez realizada la práctica de laboratorio podemos concluir en base a los resultados obtenidos que; los tres algoritmos realizados son muy eficientes a la hora de ordenar los elementos, esto debido a que no tardan más de cinco segundos en ordenar las listas hasta para 20000 elementos, para los tres posibles casos planteados, también obtenemos que a pesar de que la complejidad de Quicksort para el mejor y peor caso aunque se asemeja mucho a una complejidad cuadrática $O(n^2)$ esta no deja de ser eficiente puesto que el tiempo de ordenamiento sigue siendo muy bueno. Para finalizar podemos concluir que los tres algoritmos son muy buenos y eficientes, y si en algún momento necesitamos implementar uno de estos podremos elegir el que más nos agrade puesto que los tres son buenos algoritmos.