	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Professor: Jesus Cruz Navarro

Asignatura: Estructura de Datos y Algoritmos 2

Grupo: 01

No de Práctica(s): 01

Integrante(s): Diego Santiago Gutiérrez

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: Tercer Semestre

Fecha de entrega: 06/10/2020

Observaciones:

CALIFICACIÓN: _____

ÍNDICE:

PRACTICA 01: **ALGORITMOS DE ORDENAMIENTO PARTE 1**

OBJETIVO:

El estudiante identificará la estructura de algoritmos de ordenamiento Bubble sort y Merge sort

INTRODUCCIÓN:

Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar esa información de manera eficiente es deseable que aquella esté ordenada. Existen varios métodos para ordenar las diferentes estructuras de datos básicas.
En general los métodos de ordenamiento no son utilizados con frecuencia, en algunos casos sólo una vez. Hay métodos muy simples de implementar que son útiles en los casos en dónde el número de elementos a ordenar no es muy grande (ej, menos de 500 elementos). Por otro lado hay métodos sofisticados, más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución.

Requerimientos:
1. Programar los algoritmos de ordenamiento vistos en clase. Cada algoritmo debe estar en una función y recibir solamente un arreglo de datos. Estos son:
    a. BubbleSort Optimizado (intercambios y menos iteraciones)
    b. MergeSort

2. Desarrollar un programa para probar los algoritmos implementados anteriormente y registrar sus tiempos de ejecución con listas de tamaño n. Para probar, ambos algoritmos deben recibir el mismo conjunto de elementos (es decir, si es una lista aleatoria, los algoritmos deben recibir el mismo conjunto aleatorio). El programa debe imprimir al final n y el tiempo de ejecución de cada algoritmo. Se debe probar con listas de diferentes tamaños (n= 1000, 2000, 5000, 10000) y registrar los tiempos de ejecución para cada uno de los siguientes casos:
    a. Lista Ordenada Ascendente (Mejor Caso)
    b. Lista Ordenada Descendente (Peor Caso)
    c. Lista Aleatoria (Caso Promedio).

Dado que los tiempos pueden variar de una ejecución a otra, realice las pruebas 3 veces y obtenga el promedio de los tiempos. Nota: Para generar lista ordenada ascendentemente, agregue el valor de i del ciclo for a la lista. Para generar lista ordenada descendentemente, puede usar for i in range(max, min, -1), lo hará que i tome los valores de max hasta min (exclusivo) en cada iteración.

3. Genere una gráfica para cada caso (3 gráficas), para comparar los tiempos de ambos algoritmos en función del tamaño de la lista (n vs t, es decir, las n en el eje X y los tiempos en segundos en el eje Y).

DESARROLLO:

1.1) Merge Sort:

```python
#Creamos funcion que ira diviendo el programa
def MergeSort(arr):
    n = len(arr)
    if n > 1:
        r = n // 2

        izq = arr[:r]
        der = arr[r:]

        MergeSort(izq)
        MergeSort(der)
        Merge(arr, izq, der)
```

La función <MergeSort> dividirá el programa en partes más pequeñas, esto funciona primero calculando el tamaño del arreglo. Para ello pregunta por el tamaño del arreglo y posterior en un ciclo for verifica que tenga mas de un elemento, si es así procede a calcular la mitad del arreglo que la llama “r”.
Se crea el sub-arreglo “izq” y “der” usando como referencia la mitad del arreglo original. Realiza el mismo procedimiento en los demás sub-arreglos hasta llegar al caso base, es decir, que solo tengan un elemento. Al final estos elementos serán acomodaros por la función <merge>.

```python
#Creamos Merge que será quien acomode las listas, recibiendo izq y derecha
def Merge(arr, izq, der):
    i = 0
    j = 0
    for k in range( len(arr) ):
        if( j >= len(der) ) or (i < len(izq) and izq[i] < der[j]):
            arr[k] = izq[i]
            i = i+1
        else:
            arr[k] = der[j]
            j = j+1
```

La función Merge será quien acomode las listas, es por ello que recibe de parámetros que arreglo usará, el lado izquierdo y derecho. Empieza inicializando i y j igual a 0 para poder usarlos. En un for con variable “k” irá hasta el tamaño del arreglo que se le de a este, en este proceso realizará una comparación en donde: Si j es mayor o igual al tamaño de derecha o izquierda es menor al tamaño de izquierda (Esto quiere decir que los tamaños no se sobrepasen siendo izq y derecha). Y si izquierda es menor que derecha. Si esto es cierto, arr[k] tomará los valores de izq[i].
Si esto no se cumple, será arr[k] que tome los valores de der[j]. Siendo así que termina la llamada de la función, faltaría que terminará la recursividad de los demás programas hasta que se llegue al caso base para salir la función <MergeSort> ya con los arreglos cambiados.

1.2) Bubble Sort

```
# Creamos nuestra funcion que recibira un arreglo llamado "lista"
def bubbleSort(lista):
    #Creamos una variable global para medir  comparaciones
    global comparaciones
    #n será el tamaño de la lista que nos ayudará a interar sobre sus incisos
    n = len(lista)
    #Creamos una variable llamada intercambio para agilizar el progroama
    intercambio = False
    #El for va del inicio al final de la lista
    for i in range(1, n):
        #El for va del final al inicio
        for j in range(n-i):
            #Se va agregando un contador de cuantas veces se realiza esto
            comparaciones += 1
            #Si la lista en J es mayor a una posicion adelante
            #cambian de lugar y el intercambio ha sido cierto
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
                intercambio = True
        #De lo contrario no devuelvas nada, se queda asi
        if not intercambio:
            return

```

Creamos una función llamada <bubbleSort> la cual recibirá de parámetros una lista que iremos evaluando para acomodar los elementos de forma ascendente. Es por ello que preguntamos por su tamaño {n=len(lista)} para poder recorrer en un ciclo for del elemento 1 hasta el elemento n de la lista.
Realizamos un ciclo for que va de i hasta (n-i), es decir, hasta que acabe el primer ciclo for que es va de 1 al tamaño máximo del arreglo. Creamos una variable llamada comparaciones en la cual se va guardando cada vez que el ciclo vuelve repetirse.
Si la lista en j es mayor a su posición siguiente, realiza un cambio, de lo contrario no hay intercambio y queda así algoritmo.

1.3) Cuerpo del programa:

```
n = 10
listaNormal = []
listaMejor = []
listaPeor = []

for i in range(n):
    listaNormal.append(random.randint(0, 10))

for i in range(n):
    listaMejor.append(i)

for i in range( n, 0, -1):
    listaPeor.append(i)

comparaciones = 0

mejorA = []
mejorB = []

peorA = []
peorB = []

normalA = []
normalB = []

print ("LISTA NORMAL ES: ", listaNormal)
print ("\n MEJOR LISTA ES: ", listaMejor)
print ("\n PEOR LISTA ES: ", listaPeor)

t0 = time()
normalA = listaNormal[:]
bubbleSort(normalA)
t1 = time()
print ("\n BUBBLE ORDENAMIENTO NORMAL: ", normalA)
print ("\n Tiempo: {0:f} segundos".format(t1 - t0))
print ("\n Comparaciones:", comparaciones, "\n")
print ("n: ", n, "\n")

normalB = listaNormal[:]
t0 = time()
MergeSort(normalB)
t1 = time()
print ("\n ORDENAMIENTO MERGE NORMAL: ", normalB)
print ("\n Tiempo: {0:f} segundos".format(t1 - t0))
print ("\n Comparaciones:", comparaciones, "\n")
print ("\n n: ", n, "\n")

t0 = time()
mejorA = listaMejor[:]
bubbleSort(mejorA)
t1 = time()
print ("\n BUBBLE ORDENAMIENTO MEJOR: ", mejorA)
print ("\n Tiempo: {0:f} segundos".format(t1 - t0))
print ("\n Comparaciones:", comparaciones, "\n")
print ("n: ", n, "\n")

mejorB = listaMejor[:]
t0 = time()
MergeSort(mejorB)
t1 = time()
print ("\n ORDENAMIENTO MERGE MEJOR: ", mejorB)
print ("\n Tiempo: {0:f} segundos".format(t1 - t0))
print ("\n Comparaciones:", comparaciones, "\n")
print ("\n n: ", n, "\n")
```

```
121
122   t0 = time()
123   peorA = listaPeor[:]
124   bubbleSort(peorA)
125   t1 = time()
126   print ("\n BUBBLE ORDENAMIENTO PEOR: ", peorA)
127   print ("\n Tiempo: {0:f} segundos".format(t1 - t0))
128   print ("\n Comparaciones:", comparaciones, "\n")
129   print ("n: ", n, "\n")
130
131   peorB = listaMejor[:]
132   t0 = time()
133   MergeSort(peorB)
134   t1 = time()
135   print ("\n ORDENAMIENTO MERGE PEOR: ", peorB)
136   print ("\n Tiempo: {0:f} segundos".format(t1 - t0))
137   print ("\n Comparaciones:", comparaciones, "\n")
138   print ("\n n: ", n, "\n")
```

El cuerpo del programa se ve largo pero realmente es el más sencillo de entender, en este creamos 3 arreglos con los 3 casos que requeriremos: el mejor requiere una lista ordenada, normal requiere una lista aleatoria, el peor requiere una lista desordenada del mayor al menor.

```
57   listaNormal = []
58   listaMejor = []
59   listaPeor = []
60
61   for i in range(n):
62       listaNormal.append(random.randint(0, 10))
63
64   for i in range(n):
65       listaMejor.append(i)
66
67   for i in range( n, 0, -1):
68       listaPeor.append(i)
```

El resto del programa es llamar con merge sort y bubble sort para comparar por medio de registros de tiempo cuanto tardan en procesar estos algoritmos.

RESULTADOS:

N=1000

```
Tiempo BUBBLE MEJOR: 0.000000 segundos
Comparaciones: 500499
n:  1000

Tiempo MERGE MEJOR: 0.008014 segundos
Comparaciones: 500499

n:  1000
```

```
Tiempo BUBBLE NORMAL: 0.354089 segundos
Comparaciones: 499500
n:  1000

Tiempo MERGE NORMAL: 0.010003 segundos
Comparaciones: 499500

n:  1000
```

```
Tiempo BUBBLE PEOR: 0.440110 segundos
Comparaciones: 999999
n:  1000

Tiempo MERGE PEOR: 0.007115 segundos
Comparaciones: 999999

n:  1000
```

N=2000

```
Tiempo BUBBLE MEJOR: 0.001003 segundos
Comparaciones: 2000999
n:  2000

Tiempo MERGE MEJOR: 0.025002 segundos
Comparaciones: 2000999

n:  2000
```

```
Tiempo BUBBLE NORMAL: 1.397859 segundos
Comparaciones: 1999000
n:  2000

Tiempo MERGE NORMAL: 0.030006 segundos
Comparaciones: 1999000

n:  2000
```

```
Tiempo BUBBLE PEOR: 1.787597 segundos
Comparaciones: 3999999
n:  2000

Tiempo MERGE PEOR: 0.019009 segundos
Comparaciones: 3999999

n:  2000
```

N=5000

```
Tiempo BUBBLE MEJOR: 0.001997 segundos
Comparaciones: 12502499
n:  5000

Tiempo MERGE MEJOR: 0.054015 segundos
Comparaciones: 12502499

n:  5000
```

```
Tiempo BUBBLE NORMAL: 9.382888 segundos
Comparaciones: 12497500
n:  5000

Tiempo MERGE NORMAL: 0.058012 segundos
Comparaciones: 12497500

n:  5000
```

```
Tiempo BUBBLE PEOR: 12.907276 segundos
Comparaciones: 24999999
n:  5000

Tiempo MERGE PEOR: 0.051011 segundos
Comparaciones: 24999999

n:  5000
```

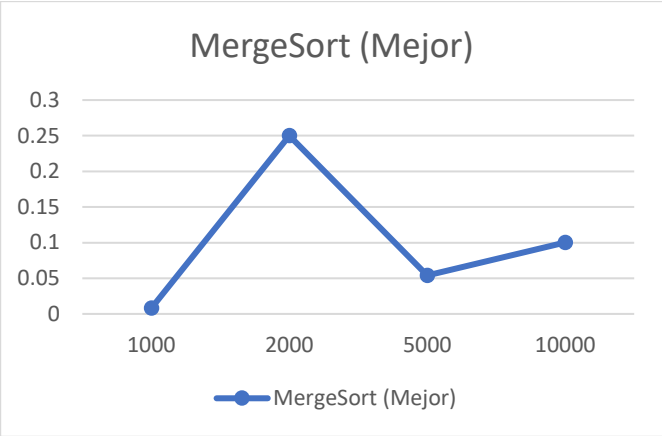
N=10000

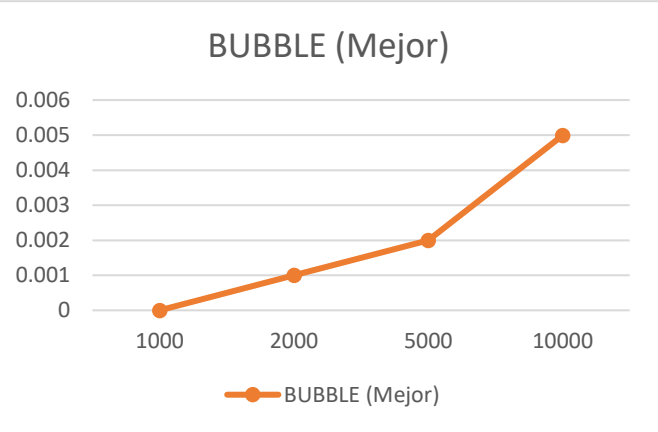
```
Tiempo BUBBLE MEJOR: 0.004999 segundos
Comparaciones: 50004999
n:  10000

Tiempo MERGE MEJOR: 0.100021 segundos
Comparaciones: 50004999

n:  10000
```

```
Tiempo BUBBLE NORMAL: 31.782430 segundos
Comparaciones: 49995000
n:  10000

Tiempo MERGE NORMAL: 0.127030 segundos
Comparaciones: 49995000

n:  10000
```

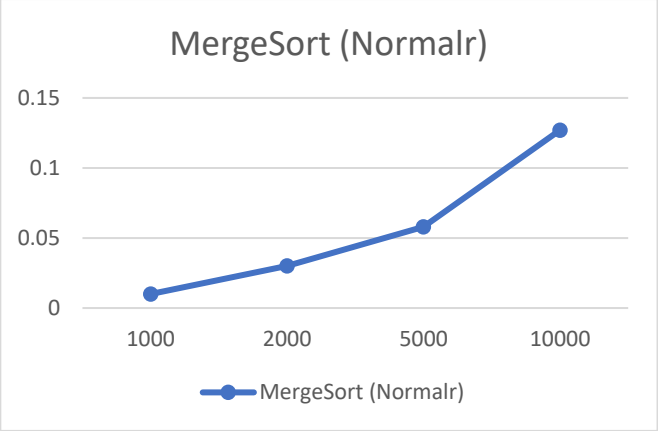
```
Tiempo BUBBLE PEOR: 43.371298 segundos
Comparaciones: 99999999
n:  10000

Tiempo MERGE PEOR: 0.104022 segundos
Comparaciones: 99999999

n:  10000
```

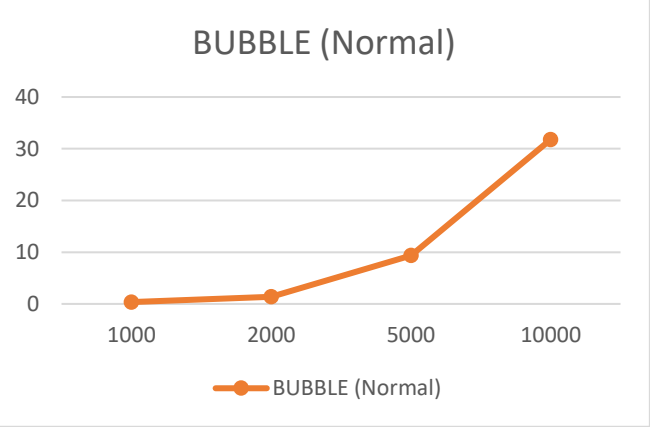
MEJOR DE LOS CASOS



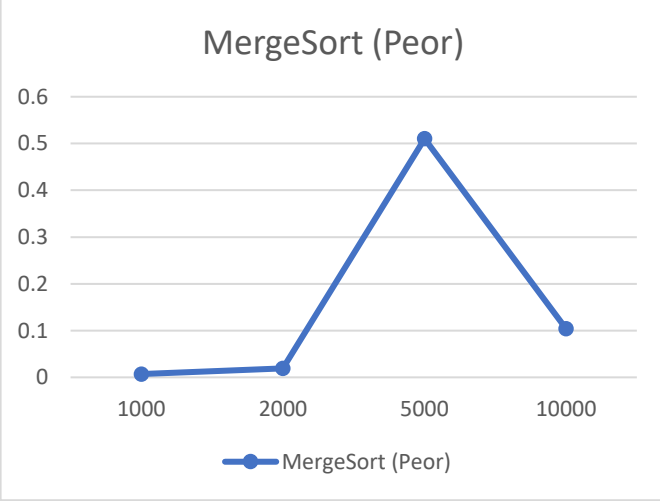


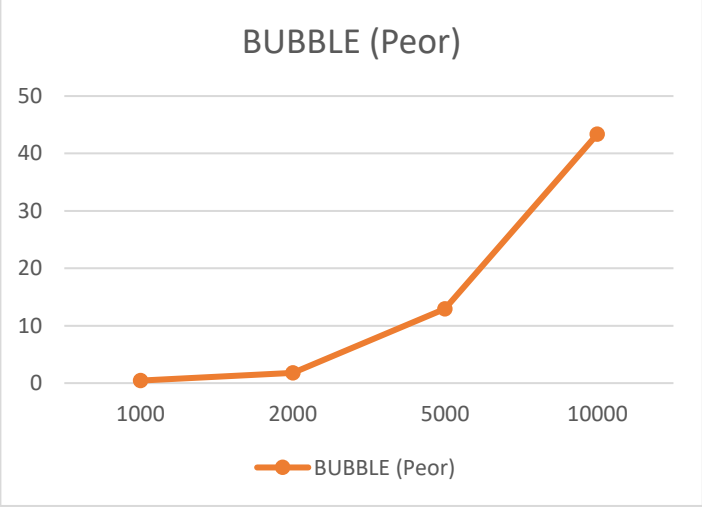
CASO NORMAL





PEOR DE LOS CASOS





X=N , Y=SEGUNDOS

CONCLUSIONES:

Es claro apreciar mediante las graficas el comportamiento de los algoritmos a mayor número de datos que estos procesen. La complejidad de los algoritmos nos ayudará a entender el por qué de estos comportamientos en cada uno de los casos.

En el mejor de los casos podemos ver como el tiempo de procesamiento es corto, siquiera llega a alcanzar el segundo tanto en Bubble como en Merge, esto es debido a que Merge se comporta como O(nLogn) Tiene la ventaja de que utiliza un tiempo proporcional a: n log (n), su desventaja radica en que se requiere de un espacio extra para el procedimiento. Este tipo de ordenamiento es útil cuando se tiene una estructura ordenada y los nuevos datos a añadir se almacenan en una estructura temporal para después agregarlos a la estructura original de manera que vuelva a quedar ordenada, es decir que de todas las complejidades, es el más rápido.

Mientras que en Bubble tiene un comportamiento muy dependiente al caso que este sea asignado, como bien es representado en la grafica BUBBLE (Peor), vemos como este es disparado en el eje de las y (segundos) debido a que alcanza una complejidad n(n-1)/2, o bien una cuadrática. En un caso promedio y el mejor, este tiene complejidad O(n), un algoritmo que puede ser más rápido en el mejor de los casos que MergeSort, pero a medida que la situación se complica para ordenar datos, este se vuelve el más ineficiente.

El análisis de los datos es una característica importante para la elaboración de algoritmos, ya que estos nos ayudarán al procesamiento de los datos, que en tiempo de ejecución representan una tarea de alta prioridad. La practica me resultó difícil, pero interesante.