	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jesús Cruz Navarro
Asignatura:	Estructuras de Datos y Algoritmos 2
Grupo:	1
No de Práctica(s):	1
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	32
Semestre:	2021-1
Fecha de entrega:	6 de octubre de 2020
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA 1: ORDENAMIENTO 1

Objetivo: El estudiante identificará la estructura de los algoritmos de ordenamiento BubbleSort y MergeSort.

1. Programar los algoritmos de ordenamiento vistos en clase. Cada algoritmo debe estar en una función y recibir solamente un arreglo de datos.

a. *BubbleSort Optimizado (intercambios y menos iteraciones)*

```python
def bubbleSort(a):
    n = len(a)

    for i in range(n - 1):
        swap = False
        for j in range(n - 1 - i):
            if a[j] > a[j + 1]:
                swap = True
                a[j], a[j + 1] = a[j + 1], a[j]

        if not swap:
            break
```

Implementación de BubbleSort

Se implementó el algoritmo analizado en clase y se agregaron las dos mejoras especificadas:

1. En cada iteración del ciclo exterior, el elemento más grande dentro de los primeros **(n – 1 – i)** elementos es “recorrido” al final del arreglo. Considerando lo anterior, podemos concluir que éste se encontrará en su lugar final, y, por lo tanto, podemos optimizar el ciclo interior de tal forma que solamente recorra los elementos necesarios, es decir, de **0** a **(n – 1 – i)**.

2. En cada iteración del ciclo exterior, se comprueba la condición **a[j] > a[j+1]** para los primeros **(n – 1 – i)** elementos. Es decir, se comprueba si existe alguna inversión entre los elementos analizados, y, de ser así, se corrige *“recorriendo la burbuja”* hasta el final del arreglo. Sin embargo, si no se realiza ninguna corrección, quiere decir que el arreglo

no contiene ninguna inversión en ese punto, por lo tanto, podemos concluir que está ordenado y podemos finalizar la ejecución del algoritmo. Lo anterior fue implementado mediante el uso de la variable **swap**.

Como ya se mencionó en el análisis previo del algoritmo, su complejidad asintótica es **$O(n^2)$.**

b. *MergeSort*

```
def mergeSort(arr):
    n = len(arr)

    if n > 1:
        r = n // 2
        izq = arr[0:r]
        der = arr[r:]
        mergeSort(izq)
        mergeSort(der)
        merge(arr, izq, der)


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
```

Implementación de MergeSort

La función **mergeSort** ordena el arreglo proporcionado. Su funcionamiento, ya analizado en clase, consiste en crear una copia de cada mitad del arreglo, ordenarlas recursivamente, y, finalmente, utilizar la subrutina **merge** para “juntarlas” en el arreglo original, de tal forma que el resultado final es una permutación ordenada de los elementos. Como ya fue analizado en clase, si el tiempo asintótico de la rutina **merge** es **O(n),** el tiempo asintótico de **mergeSort** es **O(n log(n))**. Además, cabe mencionar que el rendimiento de este algoritmo es óptimo para algoritmos de ordenamiento basados en comparación.

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).*

El programa realizado consta de distintas funciones auxiliares, las cuales se mostrarán a continuación.

Primero, se definieron algunas constantes importantes: los valores mínimo y máximo de los elementos en los arreglos a ordenar. Además, se definió una lista con los valores de n a analizar.

```
MIN_VALUE = -100000
MAX_VALUE = 100000

# Valores de n a analizar
X_ARR = [1000, 2000, 5000, 10000]
```

Constantes a utilizar en el programa

sortAndComputeTime

Esta función ordena una lista con el algoritmo proporcionado **(función sortingAlg)**, y calcula el tiempo utilizado por éste.

```python
def sortAndComputeTime(sortingAlg, arr):
    # Mantener una copia ordenada
    sortedArr = arr.copy()
    sortedArr.sort()

    # Realizar una copia para evitar mutaciones en el arreglo original
    copy = arr.copy()

    # Ordenar la lista y calcular su tiempo de ejecución
    t1 = time.time()
    sortingAlg(copy)
    t2 = time.time()

    # Comprobar que el algoritmo ordenó correctamente la lista
    assert copy == sortedArr

    # Devolver el tiempo utilizado
    return t2 - t1
```

Función sortAndComputeTime

sortAndGetTimes

Esta función ordena la misma lista con los dos algoritmos de ordenamiento analizados y retorna una tupla con los tiempos utilizados por cada uno.

```python
def sortAndGetTimes(arr):
    print("N =", len(arr))

    bubbleSortTime = sortAndComputeTime(bubbleSort.bubbleSort, arr)
    print("BubbleSort:", bubbleSortTime)

    mergeSortTime = sortAndComputeTime(mergeSort.mergeSort, arr)
    print("MergeSort:", mergeSortTime, "\n")

    return bubbleSortTime, mergeSortTime
```

Función sortAndGetTimes

computeAvgTimesArr

Esta función retorna una lista de tuplas con los tiempos promedio de ordenamiento para cada **n**, utilizando el generador de listas proporcionado. Éste último es el que determina el tipo de caso presentado (mejor, peor o promedio).

```python
def computeAvgTimesArr(listGenerator):
    # Lista con las tuplas de tiempos promedios para cada n.
    AVG_TIMES = []
    # Calcular los tiempos promedio para cada n y agregar la tupla respectiva a la lista
    for n in X_ARR:
        # Lista que contendrá 3 tuplas con los tiempos obtenidos para n
        TIMES_FOR_N = []
        # Calcular 3 veces el tiempo de los algoritmos, ordenando
        # listas diferentes en cada iteración (utilizando el generador dado)
        for i in range(3):
            arr = listGenerator(n)
            TIMES_FOR_N.append(sortAndGetTimes(arr))

        # Calcular el promedio de tiempo para cada algoritmo
        bubbleSortMean = numpy.mean([t[0] for t in TIMES_FOR_N])
        print("Tiempo promedio BubbleSort para", n, "elementos:", bubbleSortMean)
        mergeSortMean = numpy.mean([t[1] for t in TIMES_FOR_N])
        print("Tiempo promedio MergeSort para", n, "elementos:", mergeSortMean, "\n")
        # Agregar la tupla de tiempo promedio obtenida
        AVG_TIMES.append((bubbleSortMean, mergeSortMean))
    return AVG_TIMES
```

Función computeAvgTimesArr

testAndPlot

Esta función obtiene los tiempos promedios para cada **n** utilizando el generador de listas proporcionado, e imprime la gráfica con la información necesaria utilizando la librería *matplotlib*. De nuevo, **listGenerator** determina el caso a analizar, y el parámetro **title** debe contener el título del caso a mostrar en la gráfica.

```python
def testAndPlot(listGenerator, title):
    print(title, "\n")
    AVG_TIMES = computeAvgTimesArr(listGenerator)

    fig, ax = plt.subplots()
    ax.set_title(title)
    ax.set_xlabel("Tamaño del arreglo")
    ax.set_ylabel("Tiempo de ejecución")
    ax.plot(X_ARR, [y[0] for y in AVG_TIMES])
    ax.plot(X_ARR, [y[1] for y in AVG_TIMES])
    ax.legend(["BubbleSort", "MergeSort"])
```

Función testAndPlot

Funciones para cada caso

Estas funciones llevan a cabo los tests correspondientes de rendimiento para cada algoritmo, y grafican sus respectivos resultados. Nótese que, dentro de cada función, se encuentran definidas las funciones generadoras de listas para cada caso: en el caso promedio, se retorna una lista aleatoria; en el mejor caso, se retorna una lista aleatoria ordenad; y finalmente, en el peor caso se retorna una lista aleatoria ordenada de forma inversa. Como se mencionó anteriormente, éstas funciones determinan las características de los casos analizados.

```python
def mejorCaso():
    # Devuelve una lista aleatoria ordenada de tamaño n
    def listGenerator(n):
        arr = [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]
        arr.sort()
        return arr

    testAndPlot(listGenerator, "MEJOR CASO")
```

```python
def casoPromedio():
    # Devuelve una lista aleatoria de tamaño n
    def listGenerator(n):
        return [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]

    testAndPlot(listGenerator, "CASO PROMEDIO")
```

```python
def peorCaso():
    # Devuelve una lista aleatoria ordenada de forma inversa, de tamaño n
    def listGenerator(n):
        arr = [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]
        arr.sort()
        arr.reverse()
        return arr

    testAndPlot(listGenerator, "PEOR CASO")
```

Funciones para realizar los tests de cada caso

Finalmente, se ejecutan las funciones anteriores y se muestran las gráficas respectivas:

```python
casoPromedio()
mejorCaso()
peorCaso()
plt.show()
```

Ejecución de los tests y graficación

A continuación, se muestran algunas capturas de pantalla de la salida producida por la ejecución de los tres tests:

```
CASO PROMEDIO

N = 1000
BubbleSort: 0.12113595008850098
MergeSort: 0.004986763000488281

N = 1000
BubbleSort: 0.12467741966247559
MergeSort: 0.00497889518737793

N = 1000
BubbleSort: 0.11971545219421387
MergeSort: 0.0039899349212646484

Tiempo promedio BubbleSort para 1000 elementos: 0.1218429406483968
Tiempo promedio MergeSort para 1000 elementos: 0.004651864369710286

N = 2000
BubbleSort: 0.5127067565917969
MergeSort: 0.009981393814086914

N = 2000
BubbleSort: 0.49471402168273926
MergeSort: 0.008941173553466797

N = 2000
BubbleSort: 0.46280455589294434
MergeSort: 0.00893712043762207

Tiempo promedio BubbleSort para 2000 elementos: 0.49007511138916016
Tiempo promedio MergeSort para 2000 elementos: 0.00928656260172526
```

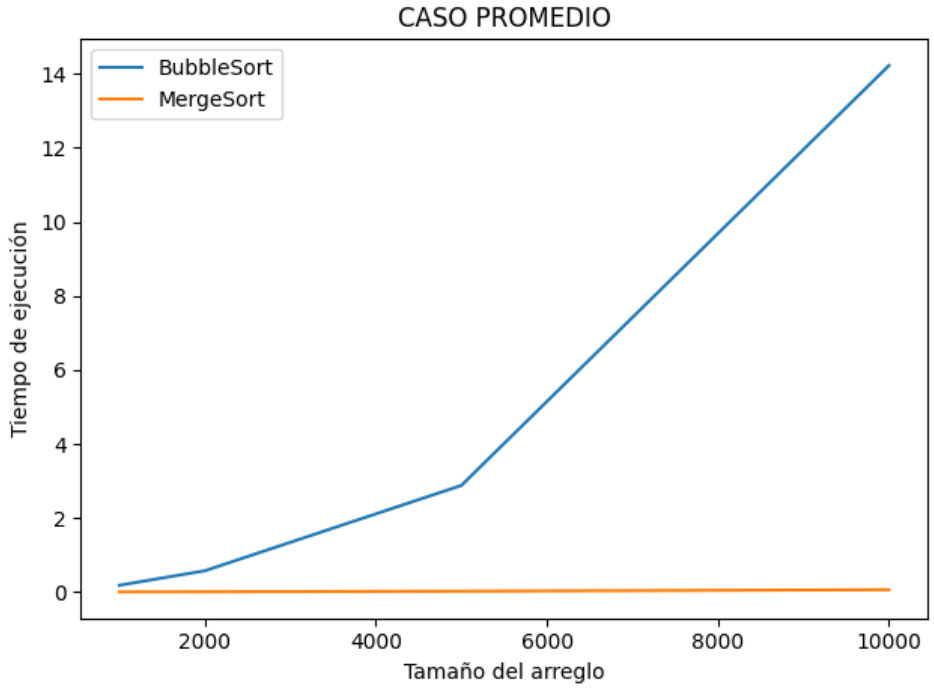
```
MergeSort: 0.05546379089355469

N = 10000
BubbleSort: 12.403560876846313
MergeSort: 0.05206894874572754

N = 10000
BubbleSort: 12.093145608901978
MergeSort: 0.054852962493896484

Tiempo promedio BubbleSort para 10000 elementos: 12.429263353347778
Tiempo promedio MergeSort para 10000 elementos: 0.054128567377726235

MEJOR CASO

N = 1000
BubbleSort: 0.0
MergeSort: 0.0029921531677246094

N = 1000
BubbleSort: 0.0
MergeSort: 0.003989219665527344

N = 1000
BubbleSort: 0.0
MergeSort: 0.003988981246948242

Tiempo promedio BubbleSort para 1000 elementos: 0.0
Tiempo promedio MergeSort para 1000 elementos: 0.003656784693400065

N = 2000
BubbleSort: 0.0
MergeSort: 0.007978677749633789
```

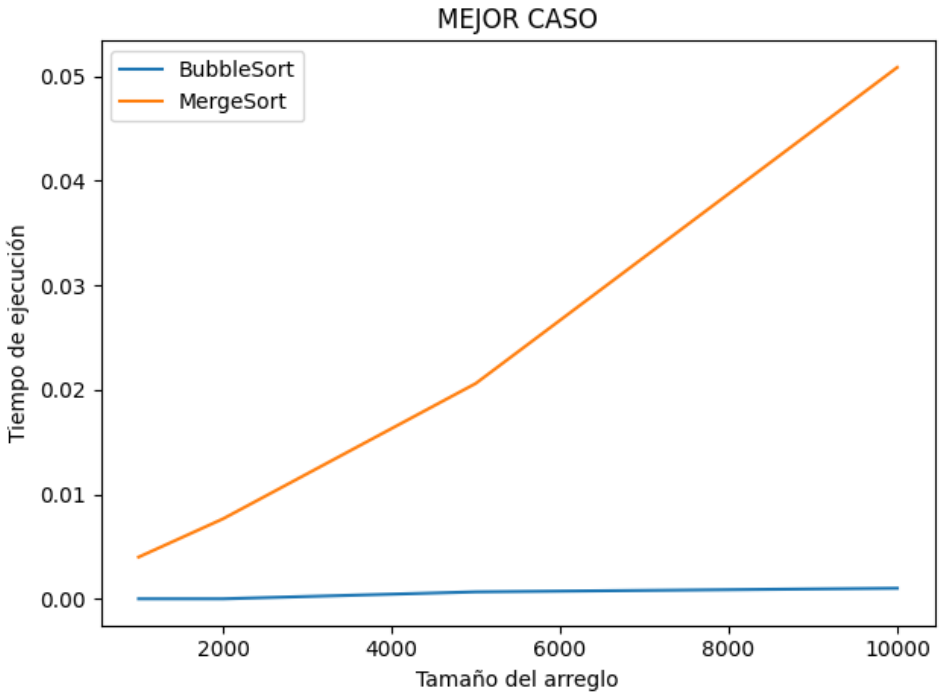
```
N = 10000
BubbleSort: 0.000997781753540039
MergeSort: 0.043883323669433594

Tiempo promedio BubbleSort para 10000 elementos: 0.0009928544362386067
Tiempo promedio MergeSort para 10000 elementos: 0.04843902587890625

PEOR CASO

N = 1000
BubbleSort: 0.1695570945739746
MergeSort: 0.0029916763305664062

N = 1000
BubbleSort: 0.16966986656188965
MergeSort: 0.004015445709228516

N = 1000
BubbleSort: 0.16898083686828613
MergeSort: 0.003988742828369141

Tiempo promedio BubbleSort para 1000 elementos: 0.1694025993347168
Tiempo promedio MergeSort para 1000 elementos: 0.003665288289388021

N = 2000
BubbleSort: 0.6840736865997314
MergeSort: 0.007938385009765625

N = 2000
BubbleSort: 0.6865274906158447
MergeSort: 0.006981372833251953
```

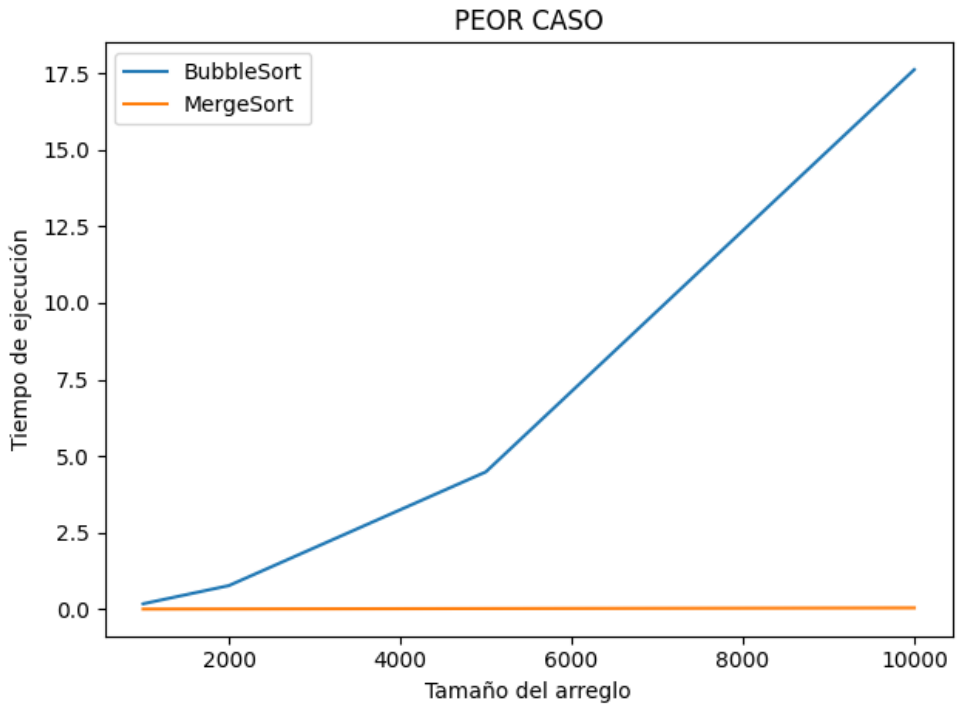
Salida del programa

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).

A continuación, se muestran las gráficas generadas por el programa anterior:







Gráficas de rendimiento para cada caso

Como se logra apreciar, la forma de las gráficas para el caso promedio y el peor caso son muy similares respecto a su forma. También se logra apreciar que el algoritmo **bubbleSort** tomó considerablemente más tiempo en el peor caso: para **n = 10000**, tomó aproximadamente **3.5** segundos más que en el caso promedio. Sin duda alguna, lo anterior comprueba la naturaleza cuadrática del algoritmo. De hecho, se logra apreciar una forma similar a la mitad derecha de una parábola.

Por otro lado, en el mejor caso, el tiempo de ejecución de **bubbleSort** fue mucho menor que el de **mergeSort,** debido a las mejoras realizadas. En realidad, en este caso **bubbleSort** únicamente realizaba un *"recorrido"* en todo el arreglo, y, al no haber corregido ninguna inversión, determinaba que el arreglo ya se encontraba ordenado y finalizaba su ejecución. En consecuencia, podemos concluir que su complejidad asintótica fue lineal (sólo para este caso). Sin embargo, en **mergeSort** fue necesario realizar las mismas operaciones, aunque la lista ya se encontrara ordenada, por lo tanto, su complejidad no se modificó.

CONCLUSIONES

BubbleSort y **MergeSort** son dos algoritmos de ordenamiento que, si bien resuelven el mismo problema, tienen complejidades asintóticas distintas: **$O(n^2)$** y **O(n log n),** respectivamente, como se determinó en clase. Como es evidente, en el análisis y diseño de algoritmos siempre es importante tratar de encontrar la solución más eficiente para el problema planteado, sin embargo, en la mayoría de las ocasiones esto no es una tarea trivial. Un claro ejemplo de lo anterior es el hecho de que el enfoque utilizado en **bubbleSort** probablemente es mucho más claro e intuitivo que el utilizado en **mergeSort**.

También es importante mencionar el hecho de haber analizado distintos casos para cada algoritmo. Por lo regular, el de mayor interés es el peor caso, ya que ningún otra entrada podría superar el tiempo de ésta. Sin embargo, cada algoritmo puede o no tener un rendimiento distinto dependiendo de su entrada.

En el caso de los algoritmos analizados en la práctica, pudimos determinar que el rendimiento de **mergeSort** fue el mismo para los tres casos. En cambio, las condiciones de las entradas influyeron considerablemente en el rendimiento de **bubbleSort**: en el peor caso, se obtuvo el rendimiento más pobre; en el mejor caso se obtuvo un rendimiento excelente, e incluso mejor que el de **mergeSort**; y, finalmente, en el caso promedio se obtuvo un rendimiento mejor que el del peor caso, pero mucho peor que el de **mergeSort**.

Como ya se mencionó, la complejidad asintótica de los algoritmos es muy importante. Durante la realización de la práctica, se logró apreciar este fenómeno claramente: el tiempo utilizado por **bubbleSort** aumentó dramáticamente hasta llegar a un máximo de **18** segundos aproximadamente, para **n = 10000.** En cambio, el tiempo de **mergeSort** no superó **un** segundo para todas las entradas, demostrando la gran importancia del diseño y análisis de algoritmos.