	Carátula para entrega de prácticas	Código	FODO-42
		Versión	01
		Página	1/1
		Sección ISO	
		Fecha de emisión	25 de junio de 2014
Secretaría/División: División de Ingeniería Eléctrica		Área/Departamento: Laboratorios de computación salas A y B	

Laboratorios de computación salas A y B

Profesor: Juan Carlos Catana Salazar

Asignatura: 1317: Estructura de datos y algoritmos II

Grupo: 8

No de Práctica(s): 1

Integrante(s) y Usuarios (HackerRank): Arrieta Ocampo Braulio Enrique (HackerRank: braulioe697)

López Santibáñez Jiménez Luis Gerardo (HackerRank: LSJGerardo)

Semestre: 2017-1

Fecha de entrega: 20 de agosto de 2016

Observaciones:

CALIFICACIÓN: _____

- **Objetivos**
    - La implementación de los algoritmos de *bubbleSort y mergeSort* utilizando como lenguaje *Phython*, para el ordenamiento de arreglos de n-números de enteros.
    - Comprobar cuál de los dos algoritmos es el más eficiente en base al comparamiento de sus tiempos de ejecución así como la cantidad de comparaciones que realiza cada uno, poniéndolos aprueba con arreglos de distintos tamaños.

- **Código en la plataforma:**
    - Algoritmo "*bubbleSort*"

Este algoritmo, básicamente funciona de la siguente manera, al inicio de la ejecución primeramente se ejecuta un ciclo for el cual se encarga de leer los datos de stdin mientras los convierte del tipo str a int, seguido de esto se manda a llamar la función *bubble* la cual es la encargada de llevar acabo el ordenamiento. En ésta a través de dos ciclos for que nos ayudan a recorrer todo el arreglo se van haciendo comparativos entre un elemento 'n' y 'n+1', en este casi si 'n > n+1' se intercambian los ejementos y se aumenta la variable comps en uno, y así con todos los elementos del arreglo hasta terminar con las iteraciones.

```
import sys                                  #Librería para el stdin
comps = 0                                   #Variable global para llevar un conteo de las comparaciones realizadas

def bubble(l):                              #Función encargada de ejecutar el algoritmo de Bubble
        global comps
        for j in range( len(l)-1 ):         #1er ciclo para iterar sobre los elementos del arreglo
                for i in range( len(l)-1-j): #2do ciclo para iterar los elementos del arreglo
                        if( l[i] > l[i+1] ): #Comparación para ver si el element 'n' es mayor que 'n+1'
                                aux = l[i]
                                l[i] = l[i+1]
                                l[i+1]= aux
                        comps+=1
        return l

#INICIO DEL LA EJECUCIÓN
for line in sys.stdin:                      #Leer los datos de stdin
   line = line.split(",")
   for cont in range(len(line)):
      line[cont] = int(line[cont]           #Convierte los datos de str a int por medio de "casteo"

bubble(line)                                #Se manda a llamar la función bubble para llevar el ordenamiento a cabo
print (comps)                               #Imprime la cantidad de comparaciones totales llevadas a cabo
```

- Algoritmo "*MergeSort*"

Funciona bajo el principio de divide y venceras, dividimos cualquier arreglo de tamaño "n" en "n" arreglos de tamaño [1], y reconstruimos el arreglo de manera que nos ahorramos la repetición de comparaciónes debido a que se ordenan en arreglos de [2], [4], [n].

```
import sys                  #Libreria importada para stdin
comparations = 0            #Se establece una variable global para llevar las comparaciones

def merge(lista1,lista2):  #Función que que compara el primer elemento de las listas y los reordena en otra auxiliar
   global comparations
   listaAux=[]
   while( len(lista1)> 0 and len(lista2) > 0):
      comparations+=1
      if( lista1[0] < lista2[0]):
         listaAux.append(lista1[0])
         lista1=lista1[1:]
      else:
         listaAux.append(lista2[0])
         lista2=lista2[1:]

   while(len(lista1)>0):
      listaAux.append(lista1[0])
      lista1=lista1[1:]

   while(len(lista2)>0):
      listaAux.append(lista2[0])
      lista2=lista2[1:]

   return listaAux

def mergeSort(lista):              #Funcion para subdividir una lista de entrada

  if( len(lista) == 1 ):
            return lista

  listaIzq = lista[ :len(lista)//2]
  listaDer = lista[ len(lista)//2:]

  listaIzq = mergeSort(listaIzq)
  listaDer = mergeSort(listaDer)

  return merge(listaIzq,listaDer)

#INICIO DEL LA EJECUCIÓN
for list in sys.stdin:              #Se lee la lista del stdin
   list = list.split(",")

for cont in range(len(list)):       #se convierte la lista a una de enteros "casteándolos"
   list[cont] = int(list[cont])

mergeSort(list)
print(comparations) #Imprime el numero de comparaciones realizadas
```

- **Código Completo (Comparación entre los algoritmos *"bubble vs merge")***

  - Algoritmo "*bubbleSort*"

```
import time              #Librería para el uso de las funciones de tiempo para poder obtener los tiempo de ejecución
import random            #Librería para el uso de la función de números aleatorios
comparations = 0         #Variable global para llevar el conteo del número de comparaciones

def BubbleSort(list):    #Función que ejecuta el algoritmo de ordenamiento de bubble
    global comparations
    for j in range(len(list)-1):
        for i in range ( len(list)-1-j ):
            comparations+=1
            if(list[i] > list[i+1] ):
                aux = list[i+1]
                list[i+1] = list[i]
                list[i] = aux
    return list


def timeElapsed(arr):            #Función para obtener los tiempos de ejecución
    global comparations
    start_time = time.time()
    #print("LISTA EN DESORDEN:  ", lista)
    #print("LISTA EN ORDEN: ", BubbleSort(arr))
    BubbleSort(arr)              #Se manda a llamar la función de bubble para ordenar la lista recibida como parámetro
    elapsed_time = time.time()  - start_time
    print("Tam_list: ", len(arr), "\tTiempo[s]: ", float("{0:.12f}".format(elapsed_time)), "\tComparaciones: ", comparations)
    comparations = 0   #Restablece el contador de las comparaciones a cero

lista = []  #Lista donde se guardaran los números generados aleatoriamente
for numero in range(1, 16):                                  #número de iteraciones (1, n+1)
    for cont in range(5*numero):                             #elementos por iteracion
        lista.append(random.randrange(-1000, 1000))          #número aleatorio en un rango
    timeElapsed(lista)                                       #Llamada a función de tiempos de ejecucion
    lista = [] #Vacia la lista
```

➢ Algoritmo "*mergeSort*"

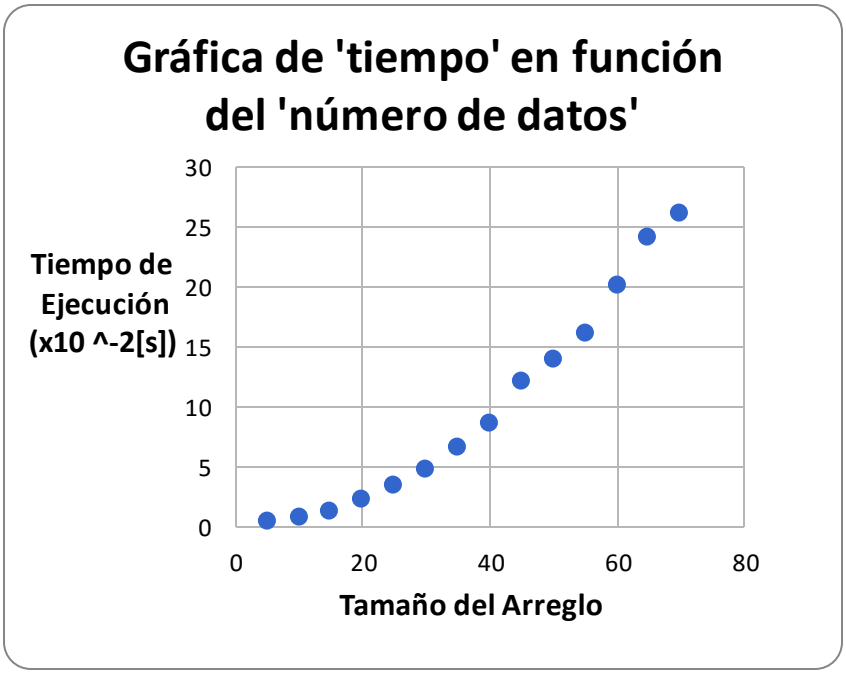
```
import time              #Librería para el uso de las funciones de tiempo para poder obtener los tiempo de ejecución
import random            #Librería para el uso de la función de números aleatorios
comparations = 0         #Variable global para llevar el conteo del número de comparaciones

def mergeList(list1,list2):                    #Función que ejecuta el algoritmo de ordenamieno de mergeSort
    global comparations
    listAux = []
    while(len(list1)>0 and len(list2)>0):
        comparations+=1
        if( list1[0] < list2[0] ):
            listAux.append(list1[0])
            list1 = list1[1:]
        else:
            listAux.append(list2[0])
            list2 = list2[1:]

    while(len(list1)>0):
        listAux.append(list1[0])
        list1 = list1[1:]

    while(len(list2)>0):
        listAux.append(list2[0])
        list2 = list2[1:]

    return listAux

def mergeSort(list):                           #Función que se encarga de subdividir una lista
    if(len(list) == 1):
        return list

    leftList  = list[:len(list)//2]
    rigthList = list[len(list)//2:]
    leftList = mergeSort(leftList)
    rigthList = mergeSort(rigthList)

    return mergeList(leftList,rigthList)

def timeElapsed(arr):                          #Función para obtener los tiempos de ejecución
    global comparations
    start_time = time.time()
    mergeSort(arr)                             #Se envia la lista a ordenar
    elapsed_time = time.time()  - start_time
    print("Tam_list: ", len(arr), "\tTiempo[s]: ", float("{0:.12f}".format(elapsed_time)), "\tComparaciones: ", comparations)
    comparations = 0

lista = []                                     #Lista donde se guardaran los números generados aleatoriamente
for numero in range(1, 16):                    #número de iteraciones (1, n+1)
    for cont in range(5*numero):               # elementos por iteracion (5,10,15,….5*número)
        lista.append(random.randrange(-1000, 1000))   #muestreo en rango
    timeElapsed(lista)                         #llamar función de tiempos de ejecucion
    lista = [] #Vacia la lista
```

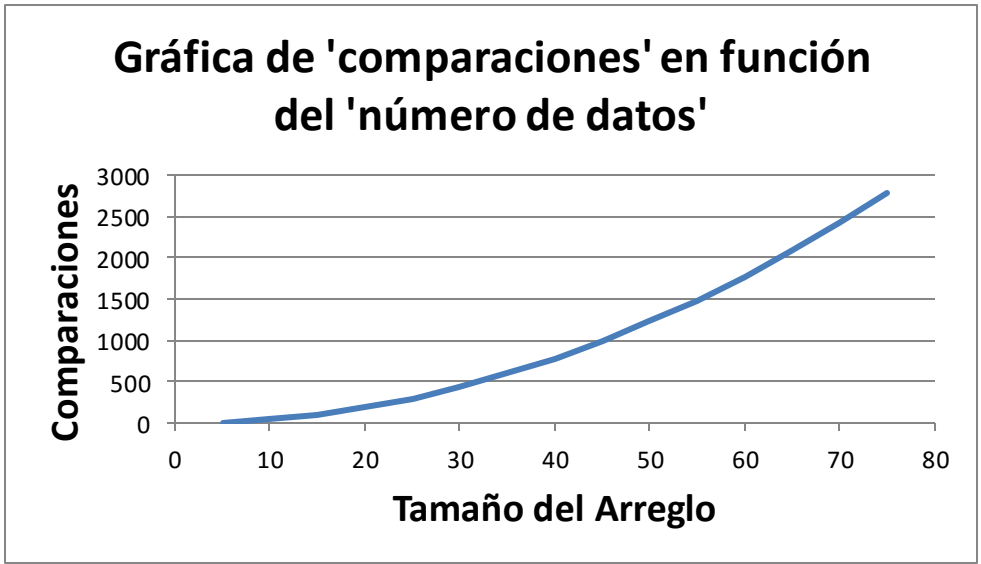
- Comparativo entre los algoritmos de “Bubble vs Merge”

Para esta prueba, en base a los códigos anteriores, se usó una función para generar un arreglo de números aleatorios que va creciendo de 5 en 5 por cada iteración que se genera, teniendo como resultado al final un total de 15 iteraciónes, de las cuales cada una nos imprime en la terminal la longitud del arreglo, los tiempos de ejecución y la cantidad de comparaciones por cada conjunto de datos.

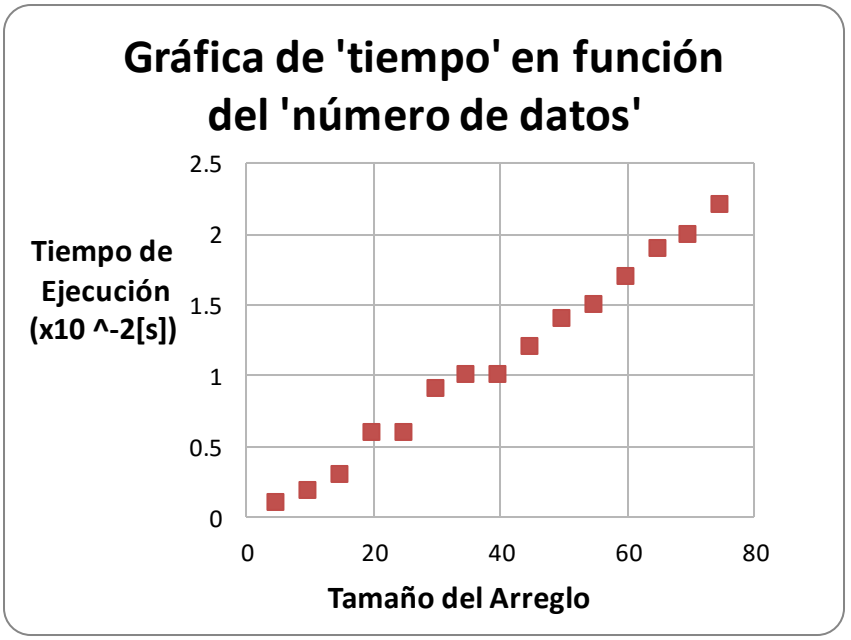
Posteriormente, con ayuda de exel, fueron graficados estos datos obteniendo los siguientes resultados.

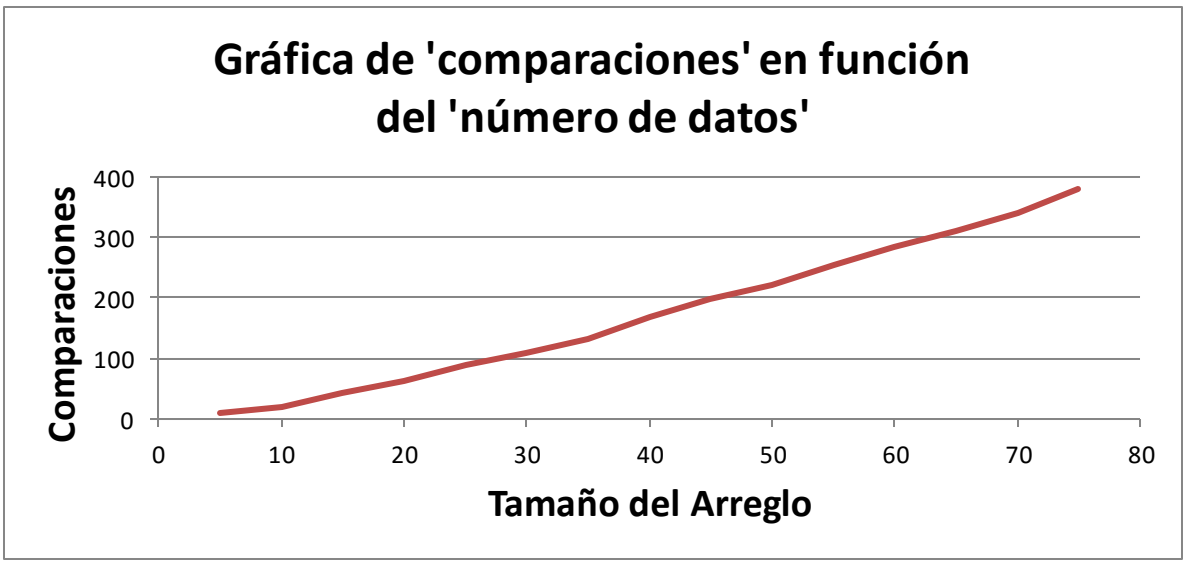
DATOS DE *BUBBLE*		
Datos	Tiempo de Ejecución (X10^-2[s])	Comparaciones
5	0.2	10
10	0.599	45
15	1.099	105
20	2.099	190
25	3.299	300
30	4.6	435
35	6.5	595
40	8.5	780
45	11.9	990
50	13.80	1225
55	16	1485
60	20	1770
65	23.899	2080
70	25.999	2415
75	30.5	2775

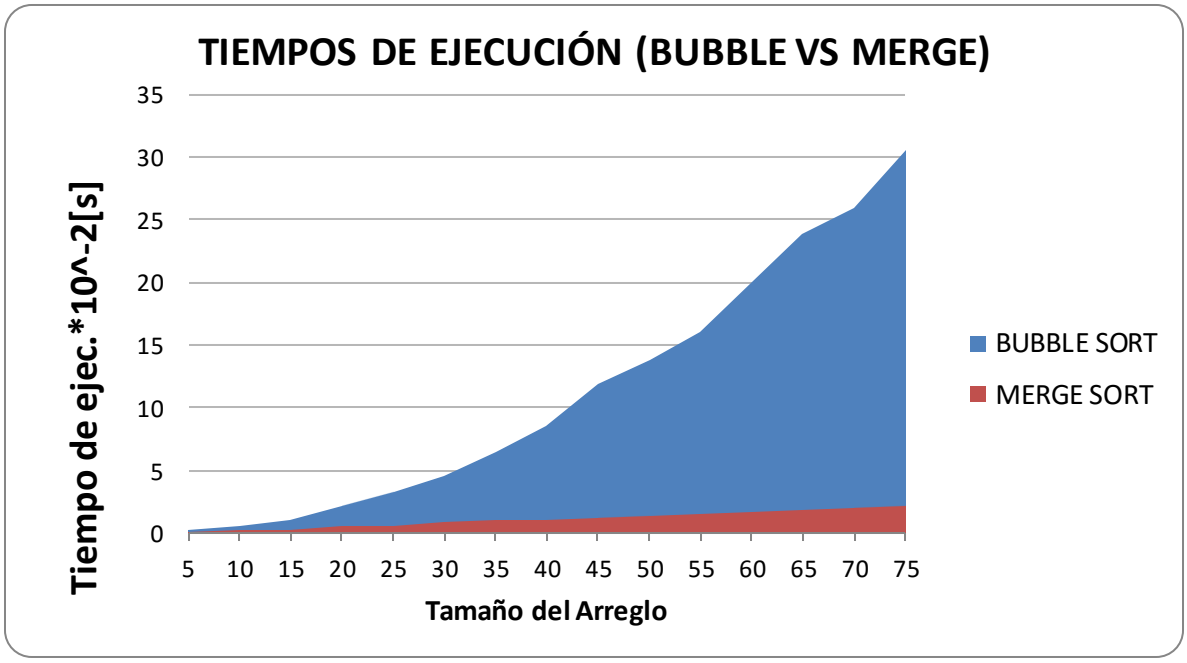


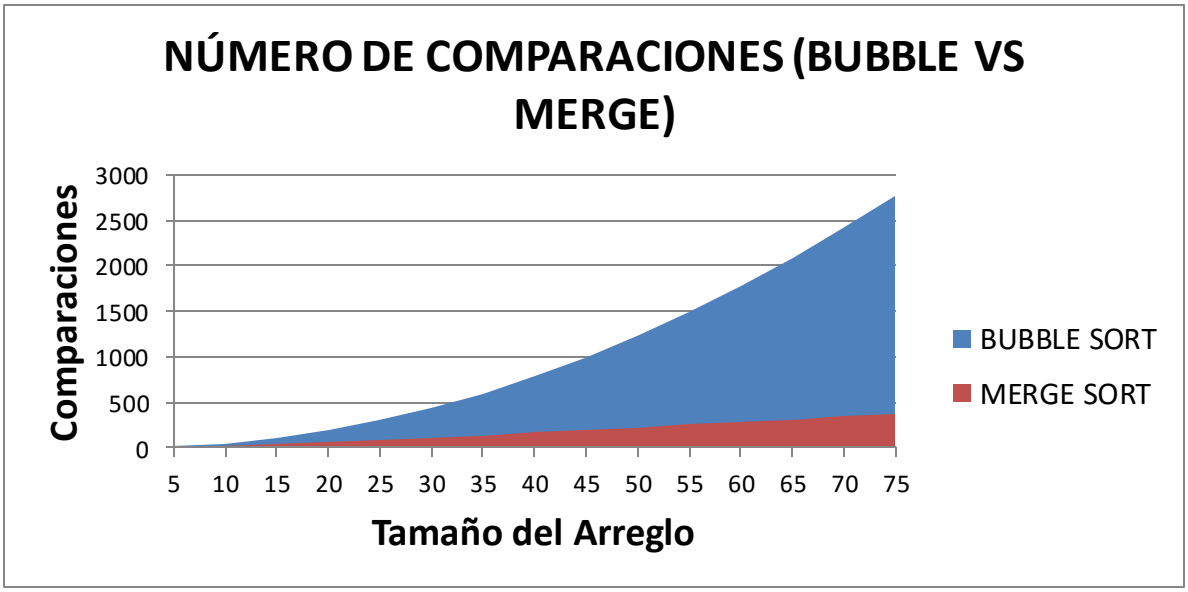


DATOS DE MERGE		
Datos	Tiempo de Ejecución (X10^-2[s])	Comparaciones
5	0.099	8
10	0.19	21
15	0.3	43
20	0.6	62
25	0.6	90
30	0.9	110
35	0.999	133
40	0.999	167
45	1.199	197
50	1.399	223
55	1.5	253
60	1.699	285
65	1.9	312
70	1.999	342
75	2.199	379









- **Conclusiones:**

  Se concluyo que el algorimo de Merge Sort tiene una mayor eficiencia que el algoritmo de Bubble Sort, ya que como se puede observar en las gráficas finales de los comparativos de tiempos de ejecución y cantidad de comparaciones, el algoritmo de Bubble para un ordenamiento de 75 elementos, presentó 2775 comparaciones en un tiempo de ejecución de 30.5x10^-2[s], en comparación con el algoritmo de Marge que presentó tan solo 379 comparaciones en un tiempo de ejecución de 2.199x10^-2[s], lo cual es bastante considerable la diferencia.