	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación
salas A y B

Profesor: Ing. Jonathan Roberto Torres Castillo

Asignatura: Estructura De Datos Y Algoritmos II

Grupo: 5

No de Práctica(s): 1

Integrante(s): Rosas Martínez Jesús Adrián

Semestre: 2018-II

Fecha de entrega: 22 de febrero 2018

Observaciones:

CALIFICACIÓN: _____

PRACTICA 1
ALGORITMOS DE ORDENAMIENTO
ESTRUCTURA DE DATOS Y ALGORITMOS II
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Profesor: Ing. Jonathan Roberto Torres Castillo

Alumno: Jesús Adrián Rosas Martínez
Correo: jesus_olimpo@outlook.com

1. Reglas de las Prácticas de Laboratorio

- Deberá respetar la estructura general del documento proporcionado para la respectiva práctica y llenar cada una de las secciones que aparecen en él.
- El desarrollo de la práctica deberá ser autentico. Aquellos alumnos que presenten los mismos cálculos, código fuente, etcétera, se les será anulada inapelablemente la práctica correspondiente con una calificación de 0.
- El día de entrega establecido deberá ser respetado por todos cada alumno, la práctica debe ser terminada parcialmente al finalizar la clase y entregada a los 8 días siguientes.
- No se recibirán informes ni actividades si no asiste a la sesión de laboratorio, por lo cual la calificación de dicha práctica será de 0.
- Deberá subir el informe en formato **PDF** al link de DropBox correspondiente a la práctica que se encuentra en la carpeta del curso. El archivo debe tener como nombre #Lista_EDAII_2018II_P#Practica.pdf. Ej: (1_EDAII_2018II_P1.pdf). A su vez debe subir los scripts de las actividades propuestas con el código que ejecuto en cada una de estas, cada script debe tener como nombre #Lista_EDAII_2018II_CODE#Actividad.py. Ej: (1_EDAII_2018II_CODE3.py).
- La inasistencia a 3 sesiones de laboratorio en el semestre será causal de reprobación de las prácticas y del curso.

2. Objetivos

- Identificar la estructura de los algoritmos de ordenamiento *Bubble Sort* y *Merge.*
- Implementar los algoritmos *Bubble Sort* y *Merge* Sort en algún lenguaje de programación para ordenar una secuencia de datos.

3. Introducción

Ordenar un grupo de datos es arreglarlos en algún orden secuencial ya sea en forma ascendente o descendente. Los algoritmos de ordenamiento de información se pueden clasificar en cuatro grupos: inserción, intercambio, selección y enumeración.

Algoritmo BubbleSort

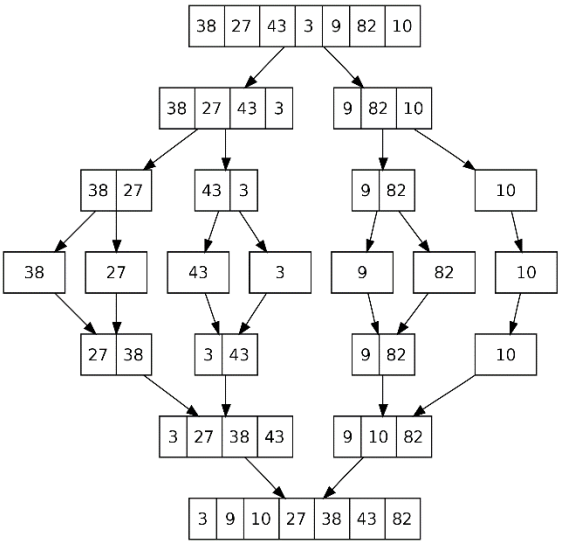
El método de ordenación por burbuja (BubbleSort en inglés) es uno de los más básicos, simples y fáciles de comprender, pero también es poco eficiente.

La técnica utilizada se denomina ordenación por burbuja u ordenación por hundimiento y es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor a otro entonces se intercambia de posición. Los valores más pequeños suben o burbujean hacia la cima o parte superior de la lista, mientras que los valores mayores se hunden en la parte inferior.

Para entender el algoritmo y como se realiza la permutación entre elementos, ver la imagen que se presenta a continuación.

Algoritmo Merge Sort

Merge Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás o también conocido como divide y conquista. Esta es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño. Consta de tres pasos: Dividir el problema, resolver cada sub-problema y combinar las soluciones obtenidas para la solución al problema original.



4. Planteamiento del Problema y Desarrollo

Actividad 1

Abajo se muestra la implementación en Python de los pseudocódigos de las funciones *bubbleSort()* y *bubbleSort2()* vistos en clase lo cuales permiten realizar el ordenamiento de forma ascendente implementando el algoritmo de ***BubbleSort***. Realizar un programa que ordene la siguiente lista {33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8} usando las dos versiones del algoritmo.

```
#Función BubbleSort
#Autor Elba Karen Sáenz García

def bubbleSort(A):
    for i in range(1,len(A)):
        for j in range(len(A)-1):
            if A[j]>A[j+1]:
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp


#Función BubbleSort Mejorada
#Autor Elba Karen Sáenz García
def bubbleSort2(A):
    bandera= True
    pasada=0
    while pasada < len(A)-1 and bandera:
        bandera=False
        for j in range(len(A)-1):
            if(A[j] > A[j+1] ):
                bandera=True
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp
        pasada = pasada+1
```

Agregar al código la impresión para conocer el número de pasadas que realiza cada función. Muestre los resultados y describa brevemente el procedimiento realizado.

1.- En la primera función (BubbleSort) el primer ciclo for controla el número de pasadas que se realizarán, esto es de 1 a 19-1, o bien, de 1 a 18. Por ello solo se agrega una línea de código que me imprima en pantalla el número de pasadas a realizar.

```
def bubbleSort(A):
    for i in range(1,len(A)): #controla el numero de pasadas (n-1)
    print("Numero de pasadas: "+str(i)+"\n")
```

```
C:\Users\jesus\Desktop>py 16_EDAII_2018_CODE1.py

Numero de pasadas: 18

[1, 1, 4, 5, 8, 9, 14, 16, 17, 18, 25, 25, 25, 32, 33, 38, 56, 76, 96]

Tiempo de ejecucion: 0.0008149016849512085 segundos
```

2.- En la segunda función (BubbleSort mejorado) cada vez que se va verificando la condición encargada de verificar si se realizó algún cambio o permutación entre valores de la lista, al final se va incrementando una variable que controla el numero de pasadas; por ello solo se agrega una línea de código que imprima en pantalla dicha variable de incremento.

```
    while pasada<len(A)-1 and bandera:
        bandera=False #Se cambia a falso, si no se
        for j in range(len(A)-1): #Numero de compar
        pasada=pasada+1
    print("\nNumero de pasadas: "+str(pasada)+"\n")
```

```
C:\Users\jesus\Desktop>py 16_EDAII_2018_CODE2.py

Numero de pasadas: 16

[1, 1, 4, 5, 8, 9, 14, 16, 17, 18, 25, 25, 25, 32, 33, 38, 56, 76, 96]

Tiempo de ejecucion: 0.004222299915855072 segundos
```

¿Qué se tiene que hacer para ordenar la lista de forma descendente? Muestre los resultados y describa brevemente el procedimiento realizado.

En ambas funciones, para ordenar la lista de forma descendente basta con cambiar la condición del if, el cual compara si un número de la lista es mayor o menor al siguiente dependiendo de la forma de ordenamiento que se requiera.

1.- Función BubbleSort

```
for i in range(1,len(A)): #contr
    for j in range(len(A)-1): #c
        if A[j]<A[j+1]:
```

```
C:\Users\jesus\Desktop>py 16_EDAII_2018_CODE1.py

Numero de pasadas: 18

[96, 76, 56, 38, 33, 32, 25, 25, 25, 18, 17, 16, 14, 9, 8, 5, 4, 1, 1]

Tiempo de ejecucion: 0.006037525162056645 segundos
```

2.- Función BubbleSort Mejorada

```
while pasada<len(A)-1 and bandera:
    bandera=False #Se cambia a falso
    for j in range(len(A)-1): #Numer
        if(A[j]<A[j+1]):
```

```
C:\Users\jesus\Desktop>py 16_EDAII_2018_CODE2.py

Numero de pasadas: 12

[96, 76, 56, 38, 33, 32, 25, 25, 25, 18, 17, 16, 14, 9, 8, 5, 4, 1, 1]

Tiempo de ejecucion: 0.004563038905910671 segundos
```

Actividad 2

A continuación, se proporciona la implementación en Python de los pseudocódigos de las funciones vistas en clase para el algoritmo ***MergeSort***. Se requiere utilizarlas para elaborar un programa que ordene las siguientes listas:

- {33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8}
- {45,25,36,12,32,14,1,4,15,19,17,13,27,28,21,0,3,5,89}

Agregar en el lugar correspondiente la línea de código de impresión, para visualizar las sub-secuencias obtenidas en la recursión.

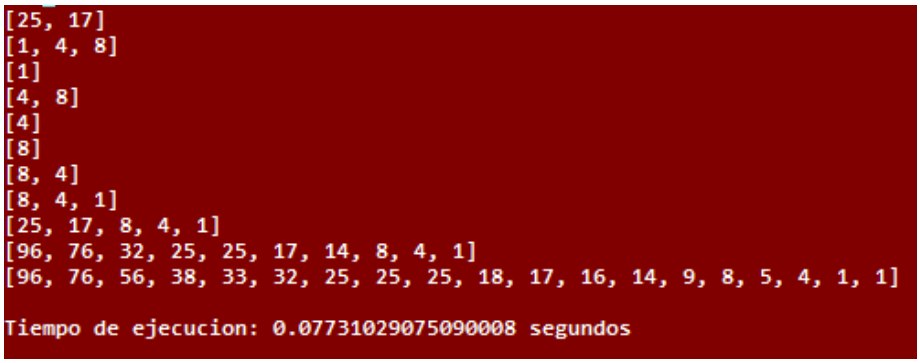
```
#MergeSort

def CrearSubArreglo(A, indIzq, indDer):
    return A[indIzq:indDer+1]

def Merge(A,p,q,r):
    Izq = CrearSubArreglo(A,p,q)
    Der = CrearSubArreglo(A,q+1,r)
    i = 0
    j = 0
    for k in range(p,r+1):
        if (j >= len(Der)) or (i < len(Izq) and Izq[i] < Der[j]):
            A[k] = Izq[i]
            i = i + 1
        else:
            A[k] = Der[j]
            j = j + 1

def MergeSort(A,p,r):
    if r - p > 0:
        q = int((p+r)/2)
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r)
```

¿Qué cambio(s) se deben hacer en el algoritmo para ordenar la lista de forma descendente? Realizar los cambios en el código, luego describir y mostrar los resultados.

Para ordenar la lista de forma descendente se debe cambiar el condicional del ciclo if ya que es el que compara cada subproblema o sublista con otra, e intercambia la posición dependiendo si una es mayor que otra.

```
if (lista1[i] > lista2[j]): #Condi
```

```
C:\Users\jesus\Desktop>py 16_EDAII_2018_CODE3.py
[33, 25, 16, 38, 56, 1, 5, 9, 18, 96, 25, 14, 76, 32, 25, 17, 1, 4, 8]
[33, 25, 16, 38, 56, 1, 5, 9, 18]
[33, 25, 16, 38]
[33, 25]
[33]
[25]
[33, 25]
[16, 38]
[16]
[38]
[38, 16]
[38, 33, 25, 16]
[56, 1, 5, 9, 18]
[56, 1]
[56]
[1]
[56, 1]
[5, 9, 18]
[5]
[9, 18]
[9]
[18]
[18, 9]
[18, 9, 5]
[56, 18, 9, 5, 1]
[56, 38, 33, 25, 18, 16, 9, 5, 1]
[96, 25, 14, 76, 32, 25, 17, 1, 4, 8]
[96, 25, 14, 76, 32]
[96, 25]
[96]
[25]
[96, 25]
[14, 76, 32]
[14]
[76, 32]
[76]
[32]
[76, 32]
[76, 32, 14]
[96, 76, 32, 25, 14]
[25, 17, 1, 4, 8]
[25, 17]
[25]
[17]
```



Actividad 3

Teniendo en cuenta la lista de números {33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8}, indique cuál de los tres algoritmos anteriores es más eficiente (tiempo de procesamiento) al ordenar los elementos de forma ascendente. Realice como mínimo 3 ejecuciones para cada algoritmo, tabule los resultados con sus respectivos promedios y luego explique brevemente al porque de los resultados.

BubbleSort	BubbleSort Mejorado	MergeSort
0.0006	0.00063	0.0003
0.00087	0.00068	0.0001
0.0009	0.0008	0.0009
0.00079	**0.0000703**	**0.00031**

El algortimo BubbleSort es el menos eficiente ya que realiza todas las comparaciones posibles en una lista de n números aun cuando algunos ya están arreglados, después le sigue el algoritmo de BubbleSort Mejorado ya que con la bandera que se pone se evita lo antes mencionado. Finalmente y como mejor opción, está el algoritmo MergeSort y es el más eficiente entre los anteriores ya que se basa en la técnica “divide y vencerás”, lo cual lo hace más eficiente cuando las listas son de números enormes.

5. Código

En esta sección se presenta el código fuente del programa que permitió cumplir los objetivos propuestos.

```
#BubbleSort
import time


def bubbleSort(A):
	for i in range(1,len(A)): #controla el numero de pasadas (n-1)
		for j in range(len(A)-1): #controla el numero de comparaciones (n-1)-1
			if A[j]<A[j+1]:
				temp=A[j]
				A[j]=A[j+1]
				A[j+1]=temp
	print("\nNumero de pasadas: "+str(i)+"\n")


A=[33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8]
tiemp_in=time.clock()
bubbleSort(A)
print(A)
tiemp_fin=time.clock()
tiemp_tot=tiemp_fin-tiemp_in
print("\nTiempo de ejecucion: "+str(tiemp_tot)+" segundos\n")
```

```
#BubbleSort
import time


def bubbleSort2(A):
	bandera=True
	pasada=0
	while pasada<len(A)-1 and bandera:
		bandera=False #Se cambia a falso, si no se hace ningun cambio dara por hecho que esta ordenado
		for j in range(len(A)-1): #Numero de comparaciones
			if(A[j]<A[j+1]):
				bandera=True #Si hace por lo menos un cambio o permutacion, el valor cambia a true
				temp=A[j]
				A[j]=A[j+1]
				A[j+1]=temp
		pasada=pasada+1
	print("\nNumero de pasadas: "+str(pasada)+"\n")


A=[33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8]
tiemp_in=time.clock()
bubbleSort2(A)
print(A)
tiemp_fin=time.clock()
tiemp_tot=tiemp_fin-tiemp_in
print("\nTiempo de ejecucion: "+str(tiemp_tot)+" segundos\n")
```

```
#MergeSort
import time
def merge(lista1, lista2): #Funcion encargada de mezclar las partes o divisiones de la lista A
	i=0
	j=0
	resultado = [] #lista vacia donde se guardara la mezcla entre lista 1 y lista2
	while(i < len(lista1) and j < len(lista2)):
		if (lista1[i] > lista2[j]): #Condicion encargada de ordenar de forma ascendente o descendente
			resultado.append(lista1[i])
			i=i+1
		else:
			resultado.append(lista2[j])
			j=j+1
	resultado = resultado + lista1[i:] #Mezcla
	resultado = resultado + lista2[j:]
	print(resultado)
	return resultado
def MergeSort(A):
	print(A)
	if len(A) < 2:
		return A
	q = len(A)//2 #Linea equivalente a q=(p+r)//2
	izq = MergeSort(A[:q]) #Division de la lista A, parte izquierda y derecha
	der = MergeSort(A[q:])
	return merge(izq, der)
A=[33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8]
tiemp_in=time.clock()
MergeSort(A)
tiemp_fin=time.clock()
tiemp_tot=tiemp_fin-tiemp_in
print("\nTiempo de ejecucion: "+str(tiemp_tot)+" segundos\n")
```

6.- Conclusión

En conclusión, los algoritmos de ordenamiento son esenciales cuando se requiere trabajar con listas de números, etcétera, encontrar el mejor método para su ordenación depende de nosotros y tenemos diversas opciones. En esta práctica se logró comprender que el método BubbleSort es un buen algortimo para ordenar datos, pero solo de un tamaño reducido. Cuando se tienen listas más grandes es mejor utilizar MergeSort y esto se puede ver al graficar la complejidad de cada algoritmo.

7. Referencias

[1] Sáenz García, Elba Karen, Guía práctica de estudio 1, Algoritmos de ordenamiento parte 1, Facultad de ingeniería, UNAM.

[2] CORMEN, Thomas, LEISERSON, Charles, et al. Introduction to Algorithms 3rd edition MA, USA The MIT Press, 2009

[3] KNUTH, Donald The Art of Computer Programming New Jersey Addison-Wesley Professional, 2011 Volumen.