	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): 2

Integrante(s): Rosas Martínez Jesús Adrián

Semestre: 2018-II

Fecha de entrega: 28 de febrero de 2018

Obervaciones:

CALIFICACIÓN: _____

PRACTICA II
ALGORITMOS DE ORDENAMIENTO PARTE 2
ESTRUCTURA DE DATOS Y ALGORITMOS II
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Profesor: Ing. Jonathan Roberto Torres Castillo

Alumno: Rosas Martínez Jesús Adrián
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

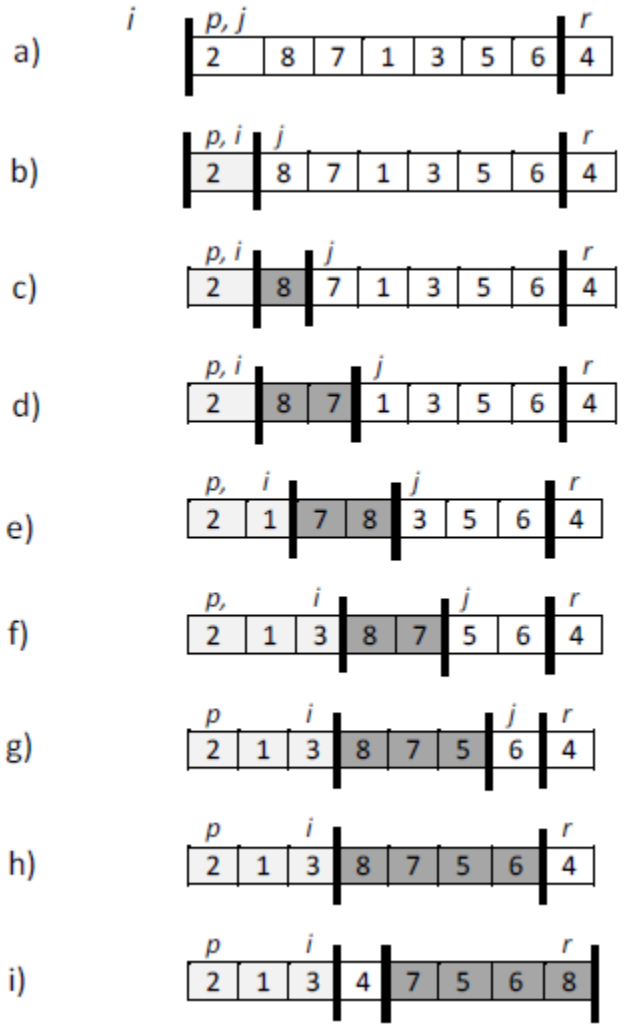
- Identificar la estructura de los algoritmos de ordenamiento *Quick Sort* y *Heap Sort.*
- Implementar los algoritmos *Quick Sort* y *Heap Sort* en algún lenguaje de programación para ordenar una secuencia de datos.

3. Introducción

Quick Sort

Este algoritmo de ordenamiento, al igual que Merge Sort, sigue el paradigma o técnica "Divide y conquista". Este algoritmo se puede explicar en tres procesos:

- Divide: Teniendo un arreglo A, se divide en 2 sub-arreglos utilizando un elemento pivote x de manera que de un lado queden todos los elementos menores o iguales a él y del otro los mayores.
- Conquista: Se basa en resolver los sub-problemas, es decir, es el proceso de ordenar los sub-arreglos de la izquierda y derecha al pivote con llamadas recursivas a la función QuickSort().
- Combina: En este proceso se combinan los sub-problemas, en este caso como ya están ordenados (conquista) no es necesario combinarlos.



El tiempo de ejecución del algoritmo depende de los particionamientos que se realizan si están balanceados o no, lo cual depende del número de elementos involucrados en esta operación.

Heap Sort

El método de ordenación HeapSort también es conocido con el nombre “montículo”. Un montículo es una estructura de datos que se puede manejar como un arreglo de objetos o

también puede ser visto como un árbol binario con raíz cuyos nodos contienen información de un conjunto ordenado. Cada nodo del árbol corresponde a un elemento del arreglo.

El algoritmo consiste de forma general en construir un heap(montículo) y después ir extrayendo el nodo que queda como raíz del árbol en sicesivas iteraciones hasta obtener el conjunto ordenado.

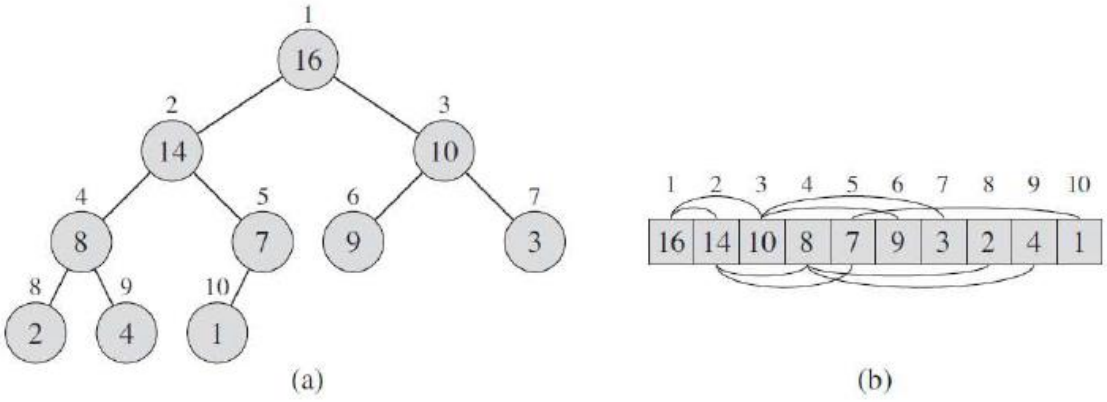


Figura 2.4 [1]

4. Planteamiento del Problema, desarrollo y resultados

Actividad 1

A partir del siguiente seudocódigo del algoritmo Quick Sort, se debe elaborar un programa en cualquier lenguaje de programación para obtener una lista ordenada de datos de forma ascendente.

Considerando que A representa el arreglo o subarreglo a ordenar, ‘p’ es el índice del primer elemento y ‘r’ la del último, se tiene el siguiente pseudocódigo:

```
Particionar(A,p,r)
Inicio
    x=A[r]
     i=p-1
     para j=p hasta r-1
 Si A[j]<=x
  i=i+1
  intercambiar A[i] con A[j]
              Fin Si
     Fin para
   intercambiar A[i+1] con A[r]
retornar i+1
Fin

QuickSort(A,p,r)
Inicio
  Si p < r entonces   // Si la lista tiene más de un elemento
   q =Particionar(A,p,r)
              QuickSort(A,p,q-1)
  QuickSort(A,q+1,r)
    Fin Si
Fin
```

Una vez elaborado el programa, responder a las siguientes preguntas.

1. ¿Qué modificaciones se tienen que hacer para ordenar la lista [33,25,16,38,56,1,5,9,18,96,25,14,76,32,25,17,1,4,8] en orden inverso? Describir, modificar el programa y mostrar los resultados.

Para ordenar la lista en orden inverso basta con cambiar la condición del IF de la función particionar ya que se comparan los elementos del arreglo o lista con el pivote; con el fin de dividir el arreglo con números menores y mayores al pivote.

```
for j in range(p,r): #se trabaja con la lista desde su primer valor ha
    if (A[j]>=x): #Condicion para que el ordenamiento sea descendente
        i=i+1
        intercambia(A,i,j)
```

```
C:\Users\jesus\Desktop>py quick2.py
[96, 76, 56, 38, 33, 32, 25, 25, 25, 18, 17, 16, 14, 9, 8, 5, 4, 1, 1]

C:\Users\jesus\Desktop>
```

2. ¿Qué se cambiaría en la función Particionar() para que en este proceso se tome como pivote el primer elemento del arreglo en lugar del ultimo? Describir, modificar el programa y mostrar los resultados.

   Python comienza a trabajar con las listas o arreglos desde el índice 0, en este caso P. Por ello para que se tome como pivote al primer elemento, se debe cambiar su posición a A[P]. Además, el índice i se debe iniciar en i=P ya que si se realiza algún cambio lo hará en i=P+1, lo cual lleva a cambiar en el ciclo FOR el rango o valores que tomará j, en este caso cambiar range(p+1,r+1) debido a que se comenzará a comparar el pivote con la siguiente posición a este hasta la última del arreglo. Finalmente se cambia afuera del FOR el parámetro i que recibe la función intercambia() por i+1 ya que es la que cambia de posición al pivote, y se debe cambiar por defecto el valor de retorno de la función Particionar por i.

```
def Particionar(A,p,r): #funcion encargada de particio
    x=A[p] #toma al primer elemento de la lista como p
    i=p
    for j in range(p+1,r+1): #se trabaja con la lista
        if (A[j]<=x): #Condicion para que el ordenamie
            i=i+1
            intercambia(A,i,j)
    intercambia(A,i,p) #cambia de posicion al pivote
    return i #nueva posicion del pivote
```

```
C:\Users\jesus\Desktop>py quick3.py
[1, 1, 4, 5, 8, 9, 14, 16, 17, 18, 25, 25, 25, 32, 33, 38, 56, 76, 96]

C:\Users\jesus\Desktop>
```

Actividad 2

A partir del siguiente pseudocódigo del algoritmo Heap Sort, se debe elaborar un programa en cualquier lenguaje de programación para obtener una lista ordenada de datos de forma ascendente.

Considerando que A representa el arreglo o subarreglo a ordenar, p es el índice del primer elemento y r la del último, se tiene el siguiente pseudocódigo:

```
MaxHeapify (A,i)
Inicio
   L= hIzq( i)
   R=hDer(i)
     Si L <TamañoHeapA y A[L]>A[i]
            posMax=L
     En otro caso
             posMax = i
     Fin Si
     Si R<TamañoHeapA y A[R]> A[posMax] entonces
             posMax =R
     Fin Si
     Si posMax ≠ i entonces
 Intercambia(A[i], A[posMax])

            MaxHeapify(A,posMax)
   Fin Si
Fin


construirHeapMaxIni( A )
Inicio
TamañoHeapA=longiudDeA
    Para i=[ longiudDeA/2], hasta 1
        MaxHeapify(A,i)
    Fin Para
Fin


OrdenacionHeapSort( A)
Inicio
       construirHeapMaxIni( A)
       Para i=longiudDeA hasta 2 hacer
           Intercambia(A[1], A[i])
           TamañoHeapA= TamañoHeapA-1;
           MaxHeapify (A,1,TamañoHeap)
Fin
```

Una vez implementado el programa, ¿Qué cambios se harían para usar un ***Heap Minimo*** (MinHeapify) en lugar de un ***Heap Máximo*** **(**MaxHeapify**)**? Describir, modificar el programa y mostrar los resultados.

Para usar un Heap Mínimo (MinHeapify), o bien, para ordenar los datos de forma descendente basta con cambiar las condiciones de los dos IF's de la función MaxHeapify ya que son las que determinan el valor o posición máxima del montículo. Al hacer dicho cambio, la función pasaría a ser MinHeapify.

```
if (L<tamanoHeap and A[L]<A[i]):
    posMax=L
else:
    posMax=i
if (R<tamanoHeap and A[R]<A[posMax]):
    posMax=R
```

```
C:\Users\jesus\Desktop>py heap2.py
[96, 76, 56, 38, 33, 32, 25, 25, 25, 18, 17, 16, 14, 9, 8, 5, 4, 1, 1]

Tiempo de ejecucion: 0.00037768820606996946 segundos
```

Actividad 3

Teniendo en cuenta la lista de números A[], indique cuál de los dos algoritmos es más eficiente (tiempo de procesamiento).

A=[54,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20

,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20
,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20,26,93,17,77,31,44,55,20]

Para determinar el algoritmo más eficiente se ejecutó 5 veces cada algoritmo:

	Quick Sort	Heap Sort
	0.035917	0.021511
	0.035697	0.022234
	0.037325	0.022073
	0.035872	0.021438
	0.038085	0.022087
Promedio	**0.0365792**	**0.0218686**

Por lo tanto, el algoritmo más eficiente es Heap Sort.

5. Código

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

```
#QuickSort

import time

def intercambia(A,i,j):
        temp=A[i]
        A[i]=A[j]
        A[j]=temp

def Particionar(A,p,r):
        x=A[r]
        i=p-1
        for j in range(p,r):
                if (A[j]<=x):
                        i=i+1
                        intercambia(A,i,j)
        intercambia(A,i+1,r)
        return i+1
```

```
def QuickSort(A,p,r):
        if (p<r):
                q=Particionar(A,p,r)
                QuickSort(A,p,q-1)
                QuickSort(A,q+1,r)

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

tiempo_in = time.clock()

QuickSort(A,0,len(A)-1)
print(A)

tiempo_fin = time.clock()
tiempo_tot = tiempo_fin - tiempo_in
print("\nTiempo de ejecucion: "+str(tiempo_tot)+" segundos\n")
```

```
#HeapSort
import time

def hIzq(i):
        return 2*i
def hDer(i):
        return 2*i+1

def intercambia(A,x,y):
        tmp=A[x]
        A[x]=A[y]
        A[y]=tmp

def MaxHeapify(A,i,tamanoHeap):
        L=hIzq(i)
        R=hDer(i)
        if (L<tamanoHeap and A[L]>A[i]):
                posMax=L
        else:
                posMax=i
        if (R<tamanoHeap and A[R]>A[posMax]):
                posMax=R
        if (posMax != i):
                intercambia(A,i,posMax)
                MaxHeapify(A,posMax,tamanoHeap)
```

```
def construirHeapMaxIni(A,tamanoHeap):
        for i in range(len(A),-1,-1):
                MaxHeapify(A,i,tamanoHeap)

def OrdenacioHeapSort(A,tamanoHeap):
        construirHeapMaxIni(A,tamanoHeap)
        for i in range(len(A)-1,0,-1):
                intercambia(A,0,i)
                tamanoHeap=tamanoHeap-1
                MaxHeapify(A,0,tamanoHeap)

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

tiempo_in = time.clock()

OrdenacioHeapSort(A,len(A))
print(A)

tiempo_fin = time.clock()
tiempo_tot = tiempo_fin-tiempo_in
print("\nTiempo de ejecucion: "+str(tiempo_tot)+" segundos\n")
```

6. Conclusiones

En conclusión, existen diferentes algoritmos para ordenar una secuencia de datos; en esta práctica se trabajó con QuickSort y HeapSort. Entre ambos algoritmos se observó mediante la práctica que ambos trabajan de forma parecida ya que se comparan valores del arreglo; en el algoritmo QuickSort se sigue la técnica divide y vencerás, por ello su complejidad es O(nlog(n)) y en el peor de los casos es O(n^2), mientras que el algoritmo HeapSort se basa en montículos o árboles en el que se sigue una serie de procedimientos para ordenar un arreglo; en este caso su complejidad es O(k+n). Por ello el algoritmo HeapSort es más eficiente. Finalmente, se cumplieron los objetivos al hacer y trabajar con ambos algoritmos en Python.

7. Referencias

[1] SÁENZ GARCÍA, Elba Karen. (2018). Guía práctica de estudio 2, Algoritmos de ordenamiento parte 2, 18 de agosto 2017, Facultad de ingeniería, UNAM. Sitio web: http://lcp02.fi-b.unam.mx/

[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.