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: 1

No de Práctica(s): 02

Integrante(s): Diego Santiago Gutiérrez

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: Tercer Semestre

Fecha de entrega: 11/10/2020

Observaciones:

CALIFICACIÓN: _____

Tabla de contenido

Objetivo **3**

Introducción 3

Desarrollo 4

InsertionSort **5**

QuickSort 5

Resultados 7-8

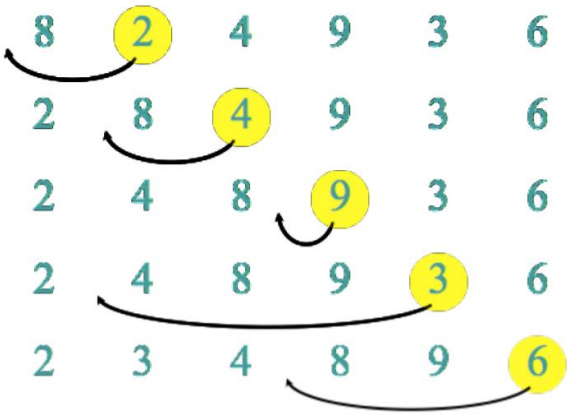
Conclusiones 9

PRACTICA 01: ALGORITMO DE ORDENAMIENTO PARTE DOS

OBJETIVO: El alumno conocerá e identificará la estructura de algoritmos de ordenamiento de Quick Sort y Insertion Sort

INTRODUCCIÓN:

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar. En este caso, nos servirán para ordenar vectores o matrices con valores asignados aleatoriamente. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada algoritmo. Este informe nos permitirá conocer más a fondo cada método distinto de ordenamiento, desde uno simple hasta el mas complejo. Se realizarán comparaciones en tiempo de ejecución, prerrequisitos de cada algoritmo, funcionalidad, alcance, etc.



Método de Insertion Sort

Método de Quick Sort

DESARROLLO:

Requerimientos:

1. Programar los algoritmos de ordenamiento vistos en clase. Cada algoritmo debe estar en una función. Estos son:
a. Quicksort (eligiendo el pivote al final)
b. InsertionSort

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= 500, 1000, 2000, 5000) y registrar los tiempos de ejecución para cada uno de los siguientes casos:

a. Lista Ordenada Ascendente
b. Lista Ordenada Descendente
c. Lista Aleatoria

3. Dado que los tiempos pueden variar de una ejecución a otra, realice las pruebas 3 veces y obtenga el promedio de los tiempos.

Nota1: Para generar lista ordenada ascendentemente, puede 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.

Nota2: Si hay problemas con el límite de recursión que usa por default Python, modifíquelo para permitir los peores casos de QuickSort.

4. 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).

1. ALGORITMOS DE ORDENAMIENTO

InsertionSort:

```python
def InsertionSort(arreglo):

    # n Será del tamaño del arreglo que reciba
    n = len(arreglo)

    # El ciclo for recorrerá de 1 al tamaño del arreglo para asegurar que todos los elementos dentro del mismo sean procesados
    for i in range(1, n):
        # se toma el valor que este dentro del arreglo en la posicion i
        val = arreglo[i]
        # J nos ayudará a separar los elementos del arreglo, en el primer caso se encontrará fuera del arreglo
        j = i - 1
        # Mientras el valor de j sea mayor o igual a cero, y el arreglo en la posición j sea mayor al valor en la posición j
        while j >= 0 and arreglo[j] > val:
            # Arreglo se recorrerá una unidad(en el lugar de valor)
            arreglo[j+1] = arreglo[j]
            # Jota decrementa en una unidad
            j = j - 1
            # El valor que se encuentre en la nueva posicion de j sera el arreglo en j
            arreglo[j+1] = val

```

Creamos nuestra función <InsertionSort> que recibirá como parámetros un arreglo, estos arreglos serán {normalA, mejorA, peorA} en los cuales veremos como se comporta el algoritmos según el caso de la lista que estemos buscando. Guardamos el valor de “n” como el tamaño del arreglo, para usarlo en un ciclo for y que vaya de 1 a “n”. Dentro del ciclo se guardará el primer valor del arreglo, “j” se reducirá en uno del valor de “i” y se abrirá un condicional while. Dentro del while es importante que j sea mayor e igual a 0 y que el valor que esta dentro del arreglo en posición “j” sea mayor al valor de “val”, esto es debido a que si el valor anterior a la posición “i” es mayor que “i”, se asignara el valor de la posición “j” de “arreglo” en la posición “j+1”, después de realizar esto, “j” retrocede un lugar de la posición que se encontraba y delante de ella regresa el valor que estaba en “i” pero un lugar atrás de la original. Este proceso lo hará hasta que la lista se encuentre ordenada por el funcionamiento del condicional.

QuickSort:

```python
def Particionar(arreglo, low, high):
    # Se saca el valor del pibote, es decir el indice hight del arreglo
    x = arreglo[high]
    # Como no se sabe si hay numeros menores, ponemos el indice afuera
    i = low - 1
    # Se recorre con un for todo el arreglo, desde el inicio hasta el final
    for j in range(low, high):
        # Si el arreglo en la posicion j es menor al pibote, lo cambia
        if (arreglo[j] <= x):
            # Si no se hace el Swapeo da problemas, esto ayuda a que se eligan los valores para el cambio
            i = i + 1
            # Se hace un Swap que dice que a[i] se cambia por a[j] y que a[j] por a[i]
            # PYTHON HACE ESTO AUTOMATICAMENTE
            arreglo[i], arreglo[j] = arreglo[j], arreglo[i]
    # Donde inicia mi parte de elementos más grandes, con el ultimo elemento, que es donde esta el piote
    arreglo[i+1], arreglo[high] = arreglo[high], arreglo[i+1]
    return i+1

# Aqui creamos la funcion quicsort que en sus parametros llevara la lista, el primer elemento del arreglo y el ultimo


def QuickSort(arreglo, iIni, iFin):
    if(iIni < iFin):
        iPiv = Particionar(arreglo, iIni, iFin)
        QuickSort(arreglo, iIni, iPiv-1)
        QuickSort(arreglo, iPiv+1, iFin)
```

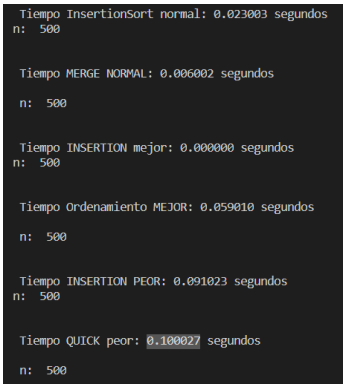
Creamos dos funciones; <QuickSort> y <Partición>, en Partición será importante darle como parámetros el arreglo (arreglo), el valor inicial (low) y el valor final (high), después de recibir estos valores, la fusión asignara a “x” (pivote) el valor final del arreglo. “i” Estará una posición atrás de el valor inicial, lo que significa que estará al inicio afuera del arreglo para evitar problemas de ejecución. Empezamos con un for que recorre del primer valor al último, en donde se abre un condicional que si el arreglo de la posición “j” es menor o igual al pivote, cambiará de posición los valores de el arreglo en “i” por el arreglo en “j” y el arreglo en “j” en la posición del arreglo “i”, es decir se hace una función Swap que Python realiza automáticamente.
Al final en el ciclo for es donde inicia la parte de elementos más grandes a partir del ultimo elemento del arreglo, terminando todo esto, retorna el valor de i+1.

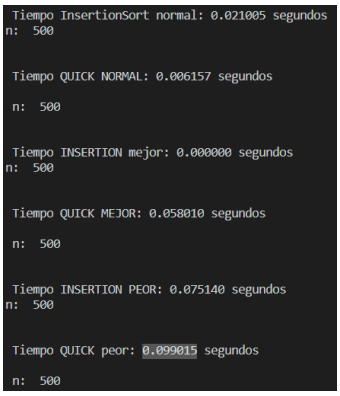
<QuickSort> es muy recursivo, ya que simplemente lo que hará es llamarse así mismo cuando vaya acomodando por medio de <Partición>, para esto se hace una comparación de si el elemento inicial es menor al final (es decir, que esta ordenada la lista), este elegirá el valor del pivote con la función <Partición>, después acomodará de los elementos antes del pivote y después de los elementos después del pivote.

2. REGISTRO DE VALORES

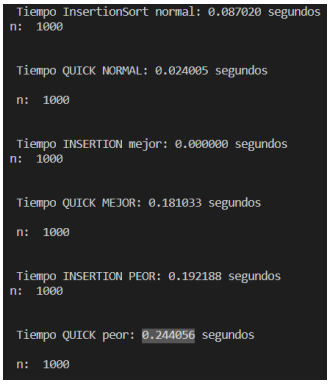
RESULTADOS:

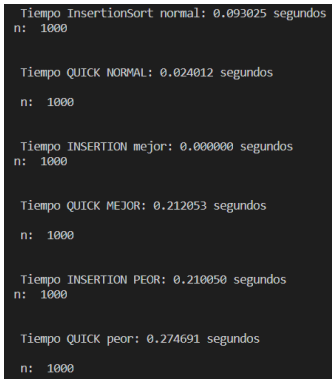
N=500

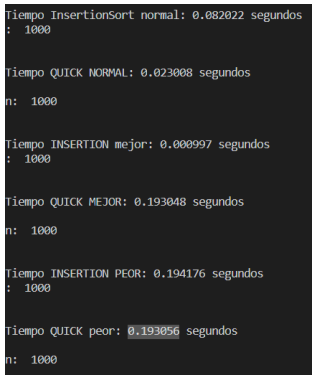




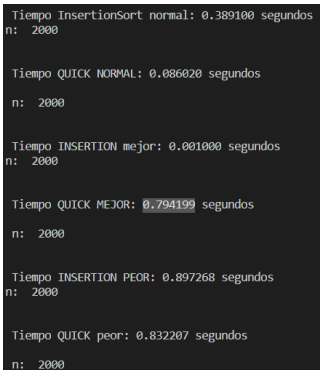
N=1000

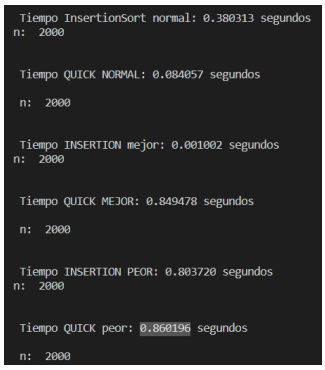


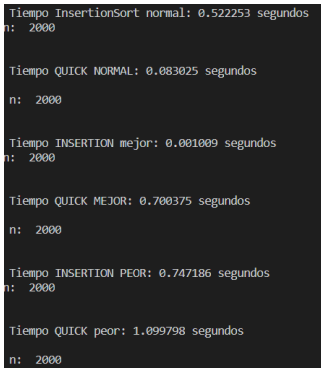




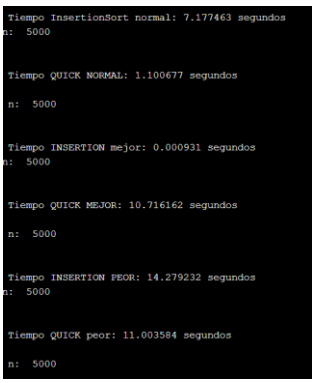
N=2000

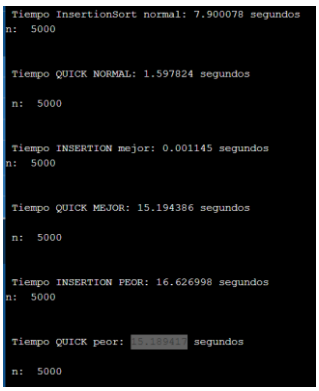


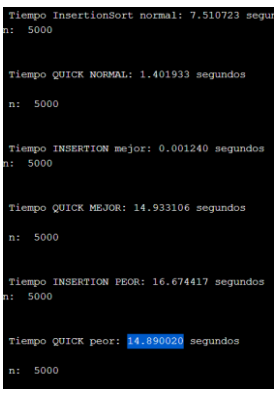




N=5000







RESULTADOS DE EJECUCIÓN:

n	P1	P2	P3	PROMEDI
500	0.023003	0.027005	0.021005	0.023671
1000	0.08702	0.093025	0.082022	0.08735567
2000	0.336086	0.3891	0.380313	0.36849967
5000	7.177463	7.900078	7.510723	7.52942133
INSERTION SORT NORMAL				

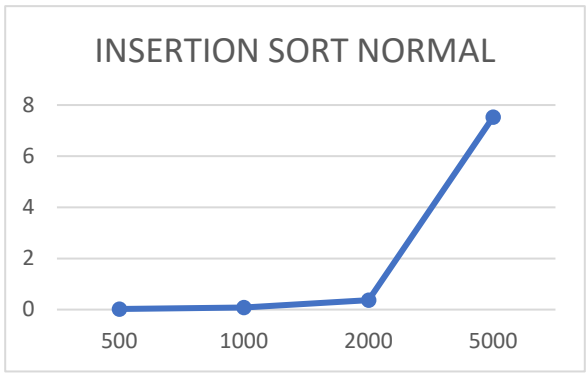
n	P1	P2	P3	PROMEDI
500	0	0	0	0
1000	0	0	0.000997	0.00033233
2000	0	0.001	0.001002	0.00066733
5000	0.000931	0.001145	0.00124	0.00110533
INSERTION MEJOR				

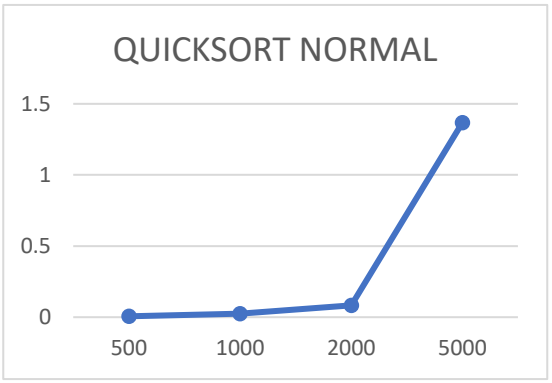
n	P1	P2	P3	PROMEDI
500	0.091023	0.081016	0.07514	0.082393
1000	0.192188	0.21005	0.194176	0.19880467
2000	0.715178	0.794199	0.80372	0.77103233
5000	14.279232	16.626998	16.674417	15.8602157
INSERTION PEOR				

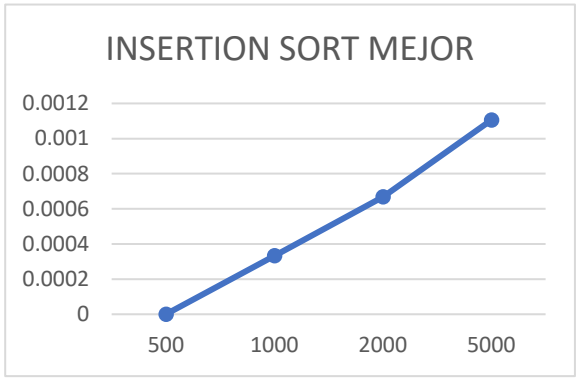
n	P1	P2	P3	PROMEDIO
500	0.006002	0.007128	0.006157	0.006429
1000	0.024005	0.024012	0.023008	0.023675
2000	0.077019	0.08602	0.084057	0.082365333
5000	1.100677	1.597824	1.401933	1.366811333
QUICKSORT NORMAL				

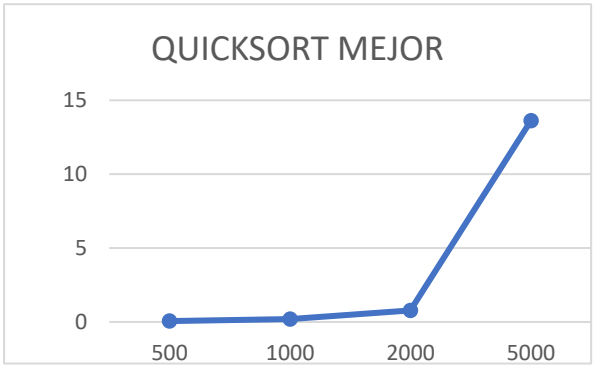
n	P1	P2	P3	PROMEDIO
500	0.05901	0.06502	0.05801	0.06068
1000	0.181033	0.212053	0.193048	0.195378
2000	0.67417	0.794199	0.849478	0.772615667
5000	10.71616	15.19439	14.93311	13.61455133
QUICKSORT MEJOR				

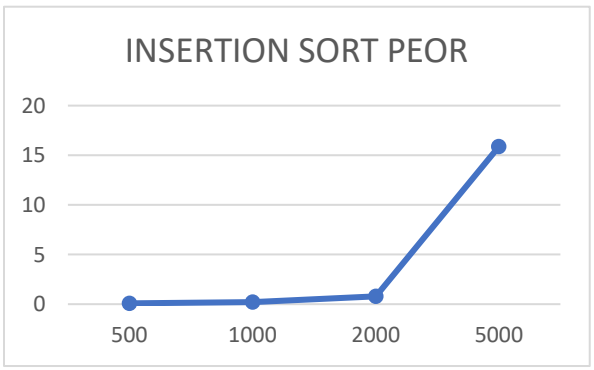
n	P1	P2	P3	PROMEDIO
500	0.100027	0.097025	0.099015	0.098689
1000	0.244056	0.274691	0.193056	0.237267667
2000	1.010779	0.832207	0.860196	0.901060667
5000	11.00358	15.18942	14.89002	13.69434033
QUICKSORT PEOR				

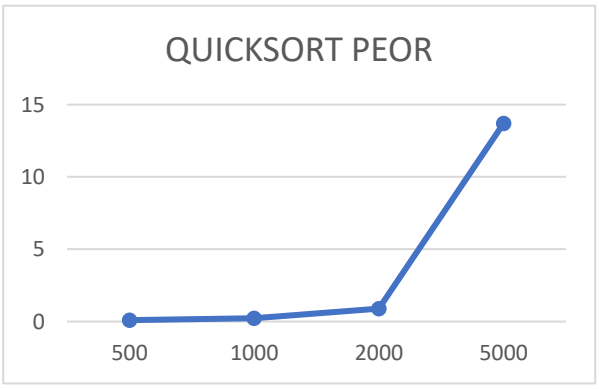












CONCLUSIONES:

InsertionSort es muy simple y fácil de implementar, además de que es eficiente en conjuntos de datos pequeños y en conjuntos de datos casi ordenados. Estable, que su tiempo de ejecución depende de la entrada: una secuencia que ya está altamente ordenada tomará menos tiempo. Desventajas podemos encontrar que es ineficiente para conjuntos grandes de elementos.

Sin lugar a duda, podemos encontrar una gran eficiencia en el algoritmo de Quicksort que en todos los casos, excepto Quick mejor, podemos encontrar un manejo de tiempo más eficiente. Es estable y además de que podemos ver que su complejidad es de (nlogn) lo que lo vuelve un algoritmo fuerte y rápido en el proceso de información.

Podemos observar que la implementación de algoritmos significa una parte importante al momento de manejar datos, ya que es algo que sin darnos cuenta, se hace diariamente y significa en computación una necesidades a priori de resolver.