INSTITUTO TECNOLÓGICO DE TIJUANA

Ing. En Sistemas Computacionales

Estructura de Datos

Proyecto Final Métodos de ordenamiento

Nava Reyes Carlos – 17212163

Hoy en día existen diferentes métodos de ordenamiento pero sin embargo no todos tienen la misma capacidad al implementarse algunos son mejores ordenando arreglos de gran tamaño que pueden no ser tan eficientes ordenando arreglos con un tamaño pequeño; En base a este problema se nos presento el realizar un proyecto donde midiéramos el tiempo de ejecución de los siguientes métodos de ordenamiento:

BubbleSort: Es uno de los algoritmos mas simples trabaja con repetidas comparaciones y cambios. Básicamente lo que hace es tomar el elemento mayor y lo recorre de posición en posición hasta ponerlo en su lugar.

QuickSort: Es uno de los algoritmos mas rápidos su tiempo de ejecución en promedio es 0(n log(n)); Cabe mencionar que es un algoritmo recursivo y si no se implementa de la manera correcta podría no tener la misma eficiencia.

MergeSort: Consiste en dividir el problema a resolver en sub-problemas del mismo tipo que a su vez se dividirán, mientras no sean suficientes pequeños o triviales; Un claro ejemplo son las matrushka que de un solo muñeco pueden salir muchos mas pequeños.

ShellSort: Su complejidad es de O(n1.2) en el mejor caso y de O(n1.25) en el caso promedio. Este algoritmo ordena los datos por inserción hace que la lista principal se divida en varias sub-listas para ir ordenando entre cierto espaciado entre los datos.

En base a estos 4 diferentes algoritmos mencionados anteriormente se realizo una tabla que nos muestra las diferentes marcas de tiempo obtenidas a lo largo de 30 pruebas con diferentes arreglos y estos son los resultados:

Merge Sort	Quick Sort	Shell Sort	Bubble Sort
0.0173690319061	0.0189599990845	0.00400686264038	0.560148954391
0.0173180103302	0.0183241367340	0.00417685508728	0.557045936584
0.0171029567719	0.0181210041046	0.00395202636719	0.560698032379
0.0186688899994	0.0195279121399	0.00400400161743	0.566720962524
0.0181329250336	0.0176191329956	0.00422310829163	0.558191061020
0.0173230171204	0.0175230503082	0.00413107872009	0.558064222336
0.0171411037445	0.0179970264435	0.00413680076599	0.558601856232
0.0188810825348	0.0191841125488	0.00497603416443	0.599734067917
0.0180900096893	0.0191779136658	0.00460505485535	0.580720901489
0.0193040370941	0.0199730396271	0.00429797172546	0.589745044708
0.0191121101379	0.0194489955902	0.00456809997559	0.611960887909
0.0199549198151	0.0230500698090	0.00488901138306	0.614174127579
0.0187640190125	0.0226860046387	0.00431704521179	0.636608123779
0.0174679756165	0.0176329612732	0.00427699089050	0.562060117722
0.0173659324646	0.0178620815277	0.00466513633728	0.565587997437
0.0178530216217	0.0202820301056	0.00680494308472	0.572509050369
0.0182130336761	0.0192608833313	0.00470280647278	0.567492961884
0.0186929702759	0.0191109180450	0.00484395027161	0.566484928131
0.0212759971619	0.0187709331512	0.00404191017151	0.576143026352
0.0180790424347	0.0178699493408	0.00447201728821	0.565778970718
0.0180079936981	0.0183310508728	0.00411486625671	0.567769050598
0.0186212062836	0.0190589427948	0.00478792190552	0.564423799515
0.0171720981598	0.0186610221863	0.00416779518127	0.576940059662
0.0184299945831	0.0186519622803	0.00397491455078	0.570313930511
0.0203490257263	0.0187349319458	0.00396084785461	0.619585990906
0.0195140838623	0.0187430381775	0.00402283668518	0.563643932343
0.0176789760590	0.0186409950256	0.00405001640320	0.570055007935
0.0188200473785	0.0180249214172	0.00434398651123	0.564635038376
0.0174930095673	0.0190119743347	0.00482892990112	0.598484039307
0.0178329944611	0.0179600715637	0.00391101837158	0.568834781647

Tiempo promedio

Merge	Quick	Shell	Bubble
0.018334317207343	0.018940035502113	0.004408494631449	0.576438562075333

Explicación del código.

Se importan las librerías necesarias que necesitaremos para algunas funciones

import random #Esta librería nos ayudara en el uso de generar listas con datos aleatorios

from time import time #Nos ayudara para tomar el tiempo de inicio y final

import sys #Con ayuda de esta librería re definiremos el máximo numero recursivo

sys.setrecursionlimit(10000) # Re-define la cantidad de veces que una función se puede llamar a si misma

Función que generara las listas aleatorias

```
def listaAleatorios(self): ## Funcion que genera nuneros aleatorios
    lista = [0]  * 5000 ## Se declara la lista y se multiplica por la cantidad de elementos que contendra para general el espacio
    for i in range(5000): ## Ciclo que genera los numeros aleatorios
        lista[i] = random.randint(1,100) ## Numeros que se generaran en un rango del 1 al 100
    return lista ## Retorna la lista generada aleatoriamente
```

Bubble Sort

```
def Bubble(self,lista):  ## Funcion que hara las respectivas comparaciones
    for i in range(1,len(lista)): ## Ciclo que recorrera el arreglo/lista
        for j in range(0,len(lista)-i):
            if lista[j] > lista[j+1]: ## Condicional que compara los datos
                aux = lista[j] ## si el dato cumple la condicion lo almacena
                lista[j] = lista[j+1] # hace que el dato mayor cambie de posicion
                lista[j+1] = aux # Asigna el valor guardado
```

Shell Sort

```
def Shell(self,lista):
    dif = len(lista) / 2 ## Establecer el tamano de la cantidad de numeros que hara el sato
    while dif >= 1: ## Permitira la ejecucion siempre y cuando la diferencia entre los datos que analizara no sea nula
        for i in range(dif,len(lista)): ## Recorre los elementos de la lista
            aux = lista[i] ## Guarda el elemento
            temp = i - dif ## GUarda el indice anterior
            while temp >= 0 and lista[temp] > aux: ##Compara los elementos y ordenara
                lista[temp + dif], lista[temp] = lista[temp], lista[temp + dif] ## Ordenada los elementos de la sublitas y los intercambia
                temp -= dif ## Decrementa los elementos
        dif /= 2 ## Reduce los espacios en los que compara formando las nuevas listas
    return lista
```

Quick Sort

```
def dividir(self,lista,menor,mayor): ## Funcion que dividira el arreglo para compararlo
    aux = (menor -1) ## Al macena el dato anterior del puntero menor
    pivote = lista[mayor] ## Asigna el pivote el dato que se encuentra en el puntero mayor
    for i in range(menor,mayor): ## Ciclo que comienza desde el menor hasta el dato mayor
        if lista[i] <= pivote: ## Compara que el dato sea menor o igual que el pivote
            aux += 1 ## Si se cumple la condicion incrementa en uno
            lista[aux],lista[i] = lista[i],lista[aux] ## Intercambia los valores para dejarlos en orden
    lista[aux+1],lista[mayor] = lista[mayor],lista[aux + 1] ## Si dentro del for no se cumple la condicion intercambia el dato mayor
    return aux+1 ## Regresa el dato de la variable aux aumentada en una para comparar el siguiente dato

def Quick(self,lista,menor,mayor): ## Funcion que ordenara los datos
    if menor < mayor: #Verifica que los datos ingresados sean menores para ordenar
        pivot = self.dividir(lista,menor,mayor) ## Crea el pivote para poder compararlos
        self.Quick(lista,menor,pivot - 1) ## Ordena la parte izquiera del pivote
        self.Quick(lista,pivot + 1, mayor) ## Ordena la parte derecha del pivote

def imprimir(self,lista): ## Funcion que imprimira los datos en pantalla
    menor = 0 ## Se le asigna el dato 0
    lis =lista
    mayor = len(lis)- 1 ## Asigna el valor en base al tamano del arreglo
    self.Quick(lis,menor,mayor) ## Manda los datos para ser ordenados
```

Merge Sort

```
def operaciones(self,izquierda,derecha): ## Funcion que ordenara los datos
    Ordena = [] ## Arreglo donde se almacenaran los datos
    while len(izquierda) != 0 and len(derecha) != 0: ## Ciclo que se ejecutara siempre y cuando el tamano de las dos variables recibidas seas diferente
        if izquierda[0] < derecha[0]: ## Compara los datos para poder ser asignados al arreglo donde se almacenan ordenados
            Ordena.append(izquierda[0]) ## Si el de la izq es menor que el de la derecha se ingresa este al arreglo
            izquierda.remove(izquierda[0]) ## Y el dato que se encuenta en izq se elimina
        else: ## De lo contrario
            Ordena.append(derecha[0]) ## Ingresara el dato que se encuentra a la derecha al arreglo que los almacena por orden
            derecha.remove(derecha[0]) ## y a si mismo lo eliminara de la derecha

    if len(izquierda) == 0: ## Si en el ciclo se cumple que el tamano de los parametros en este caso izq es igual a 0
        Ordena += derecha ## Agrupa datos
    else:
        Ordena += izquierda ## Agrupa datos
    return Ordena

def merge(self,lista):
    if len(lista) == 0 or len(lista) == 1: ## Compara que cuando el tamano del arreglo sea 1 o 0 pare para que no se mande a llamar infinitamente
        return lista ## Regresa la lista
    else:
        divide = len(lista)/2 ## Divide el arreglo
        izquierda = self.merge(lista[:divide]) ## Forma recursiva que hace que se guarde desde el 0 hasta el valor que almacene divide
        derecha = self.merge(lista[divide:])  ## Forma recursiva que hace que almacena los valor desde el valor que tiene divide hasta el final
    return self.operaciones(izquierda,derecha) ## Manda a llamar a la funcion que los ordenara
```

Una vez que se codificaron los diferentes algoritmos de ordenamiento se mandaron a llamar con una variable que se le asigno la clase principal para poder hacer uso de las diferentes funciones que contienen a los algoritmos. Se mando a llamar a la función que contiene los datos aleatorios y esta misma variable se igualo a 3 variables mas para que todos los algoritmos de ordenamiento ordenaran el mismo grupo de datos.

```
up = Metodos()
lista = up.listaAleatorios()
ShellAr = QuickAr = MergeAr = lista
```

Una vez que se realizo todos lo anterior se procedió a mandar a llamar las diferentes funciones con el mismo formato y sus respectivas funciones y mensajes.

```
print("\n--> Merge sort <--\n") ## Imprime el nombre del algoritmo que esta corriendo
start = time() ## Almacena el tiempo de inicio
up.merge(MergeAr) ## Manda a llamar el algoritmo que ordenara los datos
end = time() ## Almacena el tiempo con el que finalizo el algoritmo
total = end - start ## Con una resta se saca el tiempo total de ejecucion
print("Tiempo de ejecucion " + str(total) + " Segundos\n") ## Imprime en pantalla el tiempo de ejecucion
```