Suarez Vázquez Brandon Abraham 17211566

Estructura de datos

Proyecto: Algoritmos de ordenamiento

El propósito de este proyecto es evaluar el tiempo en que los algoritmos ordenan el mismo arreglo de x dimensión, midiendo los tiempos y colocándolos en una tabla para después obtener el promedio.

Algoritmos a evaluar:

Burbuja

QuickSort

ShellSort

MergeSort

De las líneas 13-39 se explica el funcionamiento del método burbuja:

```
13 #definimos nuestro metodo de ordenamiento burbuja, que recibe como parametros
14 #la lista y un puntero x
15 def metodoburbuja(lista, x):
16     #usamos la siguiente condicion para verificar si la variable x es menor
17     #que la longitud del arreglo
18     if x<len(lista):
19         #si es menor, incrementa su valor en 1
20         x+=1
21         for i in range(len(lista)-x):
22             #creamos un ciclo for que va desde 0 hasta la longitud del arreglo -1
23             #checamos si el valor en el indice actual es mayor al valor en el indice siguiente
24             if(lista[i]>lista[i+1]):
25                 #de ser asi, se crea una variable auxuliar t, que nos
26                 #servira para guardar momentaneamente el dato
27                 t=lista[i]
28                 #luego, el dato que se encuentra adelante se mueve atras al ser menor
29                 lista[i]=lista[i+1]
30                 #ahora, sustituimos el valor de adelante con el valor que se
31                 #encontraba atras, ya que este es mayor
32                 lista[i+1]=t
33         #usamos recursividad para volver a llamar al metodo el numero
34         #de veces necesario hasta terminar de ordenar la lista
35         metodoburbuja(lista,x)
36     else:
37         #cuando la variable x ya no sea menor que la longitud del arreglo
38         #se entiende que se ha terminado de ordenar, se imprime  el siguiete mensaje
39         print("Lista ordenada")
40
```

Desde la 35-74 se explica el funcionamiento del método Quicksort:

```
35 #definimos nuestro metodo quicksort, el cual recibe una lista como parametro
36 def QuickSortez(lista):
37     #comparamos si la longitud de la lista es menor o igual a 1, lo que quiere
38     #decir que se ha logrado partir y ordenar la lista en arreglos de hasta
39     #dimensión 1, entonces, devuelve la lista
40     if len(lista)<=1:
41         return(lista)
42     #creamos 3 arreglos, uno para los menores que el pivote, para los iguales
43     #y para los mayores que el pivote
44     menorespiv=[]
45     igualespiv=[]
46     mayorespiv=[]
47     #definimos como pivote el numero que se encuentra justo a la mitad
48     #de la lista
49     pivote=lista[len(lista)//2]
50 
51     #iniciamos un ciclo for, que irá desde i hasta la longitud de la lista
52     for i in lista:
53         #si el dato que se encuentra en i es menor que el pivote, entonces
54         #lo agregamos al arreglo de menores
55         if i<pivote:
56             menorespiv.append(i)
57         #si el dato en la posicion de i es igual al pivote, lo agregamos
58         #al arreglo de iguales
59         elif i==pivote:
60             igualespiv.append(i)
61         #si no es ni menor ni igual, se entiende que es mayor, entonces
62         #lo agregamos al arreglo de mayores
63         else:
64             mayorespiv.append(i)
65     #retornaremos la concanetación de los arreglos
66     #notese que se usa la recursividad en los arreglos de
67     #menores y mayores para que en las posteriores ejecuciones
68     #del metodo quicksort sobre estas listas, se repita el proceso de seleccionar
69     #los mayores, menores e iguales y ponerlos a su vez en respectivas listas
70     #esto se seguirá ejecutando hasta que la longitud de los arreglos sea <=1
71     #entonces se irán retornando las listas arregladas por cada ejecución del
72     #metodo quicksort hasta llegar a la primera ejecución, devolviendo entonces
73     #la lista ordenada
74     return QuickSortez(menorespiv)+igualespiv+QuickSortez(mayorespiv);
75 
76 
77 
```

A partir de la línea 79-110 se explica el funcionamiento del ShellSort:

```
79 #definimos nuestro metodo shellsort que recibe como parametro un arreglo
80 def ShellSort(lista):
81     #definimos de cuanto será nuestro salto entre los datos del arreglo
82     #dividimos la longitud entre 2 tomando solo el entero
83     salto = len(lista) // 2
84 
85     #creamos un ciclo while con la condición de: mientras el salto
86     #sea mayor que 0
87     while salto > 0:
88         #se ejecutará este ciclo for, en el rango de i hasta la longitud de la
89         #lista
90         for i in range(salto, len(lista)):
91             #guardamos el numero que se encuentra en la posicion de i
92             #en una variable e igualamos el puntero z a la posición de i
93             num = lista[i]
94             z = i
95             #creamos un ciclo anidado while con las siguientes condiciones:
96             #mientras el valor de z sea mayor o igual al valor del salto
97             #y el dato que esta en la posicion de z menos el salto mayor
98             #que el numero que guardamos en la variable num, osea el numero que
99             #se ecuentra en la posicion del salto, entonces los datos se cabian
100            while z >= salto and lista[z - salto] > num:
101                lista[z] = lista[z - salto]
102                #disminuiremos el valor de z en la cantidad da salto para
103                #ir avanzando en el arreglo y cambiar las posiciones cuando
104                #la condición se cumpla
105                z -= salto
106            #a la posición de z en la lista, le asignamos el valor de num
107            lista[z] = num
108        #dividiremos el valor del salto entre 2 y obtendremos el valor
109        #entero solamente despues de cada iteración
110        salto //= 2
111 
112
```

Entre las líneas 115-196 se explica el funcionamiento del MergeSort:

```
115 #definimos el metodo MergeSort el cual recibe una lista
116 def MergeSort(lista):
117     #de entrada, comparamos si la longitud de la lista es mayor a 1
118     #esto servirá poesteriormente cuando se llame de manera recursiva al
119     #metodo enviandole listas divididas cada vez mas pequeñas
120     if len(lista) >1:
121         #calculamos el punto medio al dividor la longitud
122         #del arreglo entre 2 y tomando solo el entero
123         mid = len(lista)//2
124         #creamos 2 sublistas que abarcaran desde el inicio hasta el punto medio
125         #osea, la lista izquierda y otra desde el punto medio hasta
126         #el final de la lista,osea la lista derecha
127         I = lista[:mid]
128         D = lista[mid:]
129         #entonces, usamos la llamada recursiva al metodo al cual enviaremos
130         #las listas divididas, que a su vez, se dividirán en más hasta que la
131         #longitud sea 1
132         MergeSort(I)
133         MergeSort(D)
134 
135         #utilizamos 3 punteros inicializados en 0
136         x = 0
137         y = 0
138         z = 0
139 
140         #en esta parte, moveremos los datos de ambas sublistas
141         #a la lista original, pero ordenandolos en el proceso
142         #declaramos un ciclo while con doble condición:
143         #mientras el valor de x sea menor que la longitud de la sublista
144         #izuiqerda y el valor de y menor que la longitud de la sublista
145         #derecha entonces:
146         while x < len(I) and y < len(D):
147             #realiza la comparación:
148             #si el elemento que esta en la primera posición de
149             #la sublista izquierda es menor que el elemento que
150             #esta en la primera posición de la sublista derecha
151             #entonces en la posicion de z de la lista original
152             #almacena el dato de la sublista izquierda, al ser menor
153             if I[x] < D[y]:
154                 lista[z] = I[x]
155                 #entonces incrementamos el valor del puntero x
156                 #para seguir recorriendo elementos hasta vacear la lista
157                 x+=1
158             #si no es menor, entonces se entiende que el elemento de la
159             #sublista derecha es menor que el de la sublista izquierda
160             #entonces almacena ese elemento en la lista original
161             else:
162                 lista[z] = D[y]
163                 #incrementamos el valor de y para seguir recorriendo y
164                 #moviendo elementos a la lista original
165                 y+=1
166             #despues de cada comparación y movimiento de de elementos
167             #aumentamos el valor del puntero de la lista original
168             #para que el elemento siguiente quede insertado enseguida
```

```
            y+=1
        #despues de cada comparación y movimiento de de elementos
        #aumentamos el valor del puntero de la lista original
        #para que el elemento siguiente quede insertado enseguida
        #del ultimo ingresado
        z+=1

        #en esta parte, verificamos que ningun elemento haya quedado
        #atrás sin mover a la lista original

    #el ciclo while con la condición:
    while x < len(I):
    #mientras el valor de x sea menor que la longitud de la sublista izquierda
    #entonces en la posición de z en la lista original, se almacenará
    #el dato que se encuentra en la posición de x en la sublista izquierda
        lista[z] = I[x]
        #despues de cada inserción de elementos de la sublista izquierda
        #a la lista original, incrementamos el valor de los punteros en 1
        #para proseguir con con el ciclo hasta que este deje de cumplirse
        x+=1
        z+=1
    #el ciclo while con la condición:
    while y < len(D):
        #mientras el valor de y sea menor que la longitud de la sublista derecha
        #entonces en la posición de z en la lista original, se almacenara
        #el dato que se encuentra en la posición de y en la sublista derecha
        lista[z] = D[y]
        #despues de cada inserción de elementos de la sublista derecha
        #a la lista original, incrementamos el valor de los punteros en 1
        #para proseguir con con el ciclo hasta que este deje de cumplirse
        y+=1
        z+=1
```

A partir de la línea 200-227 se explica la forma en que se llena la lista:

```
#creamos nuestra lista vacía
lista=[]

#definimos nuestro metodo para la inserción de numeros aleatorios, el cual
#recibe como parametro la lista y el contador
def insercion(lista,c):
    global listab
    global listaq
    global listas
    global listam
    #si el contador sea menor que 50
    if (c<100):
        #crea un numero aleatorio entre 0 y 100 y lo agrega a la lista
        #incrementa el valor del contador en 1 y se usa recursividad
        #para llamar al metodo hasta que que se hayan insertado los 50 elementos
        e=randint(0,1000)
        lista.append(e)
        c+=1
        insercion(lista,c)
    #cuando ya no se cumple la condicion, entonces imprimimos este mensaje
    #y creamos diferentes listas (una para cada metodo de ordenamiento)
    #asignamos a cada una de las listas los datos de la original
    else:
        print("Insercion terminada")
        listab=lista
        listaq=lista
        listas=lista
        listam=lista
```

De la línea 230-284 se crea el menú y se ejecutan los métodos de ordenamiento según lo desee el usuario:

```
230 def menu():
231     print("Crear una nueva lista aleatoria:1")
232     print("Metodo Burbuja:                 2")
233     print("Metodo QuickSort:               3")
234     print("Metodo ShellSort:               4")
235     print("Metodo MergeSort:               5")
236     opc=int(input());
237     if (opc==1):
238         lista=[]
239         c=0
240         insercion(lista,c)
241         menu();
242     if (opc==2):
243         i=0
244         tib=time.clock()
245         metodoburbuja(listab,i);
246         tfb=time.clock()-tib
247         #tb=tfb-tib
248         print("Lista ordenada por metodo burbuja")
249         print(listab)
250         print("Tiempo tomado para ordenamiento")
251         print(tfb)
252         menu();
253     if (opc==3):
254         tiq=time.clock()
255         listaquick=QuickSortez(listaq);
256         tfq=time.clock()-tiq
257         #tq=(tfq-tiq)
258         print("Lista ordenada por Quicksort")
259         print(listaquick)
260         print("Tiempo tomado para ordenamiento")
261         print(tfq)
262         menu();
263     if (opc==4):
264         tis=time.clock()
265         ShellSort(listas);
266         tfs=time.clock()-tis
267         #ts=(tfs-tis)
268         print("Lista ordenada por Shellsort")
269         print(listas)
270         print("Tiempo tomado para ordenamiento")
271         print(tfs)
272         menu();
273     if (opc==5):
274         tim=time.clock()
275         MergeSort(listam);
276         tfm=time.clock()-tim
277         #tm=(tfm-tim)
278         print("Lista ordenada por Mergesort")
279         print(listam)
280         print("Tiempo tomado para ordenamiento")
281         print(tfm)
282         menu();
283
284 menu();
```

Primero, damos las diferentes opciones al usuario, entonces, capturamos su decisión y la comparamos con las posibles alternativas:

Cuando el usuario elige crear una lista, primero definimos una como vacía e igualamos el contador a 0, entonces llamamos al método de inserción y usamos recursividad para volver al menú.

Entonces, el usuario podrá elegir entre los métodos de ordenamiento para ejecutar, antes de la ejecución de cada método, usamos la función time para medir el tiempo de inicio del algoritmo, posteriormente ejecutamos el mismo mandándole la lista que le corresponde, después de su ejecución hacemos uso de nuevo de la función time para medir el tiempo final de ejecución, para determinar el tiempo definitivo restamos el tiempo inicial menos el final.

Mostramos entonces el arreglo ordenado por el método y el tiempo que este tardo en ejecutarlo.

Para dar inicio a la ejecución de nuestro programa, simplemente ejecutamos el menú.

Después de ejecutar cada método de ordenamiento 30 veces se obtuvieron los siguientes resultados:

Algoritmo/Intento	Burbuja	QuickSort	ShellSort	MergeSort
1	0.087072	0.000998	0.002014	0.003015
2	0.088072	0.000988	0.001011	0.003005
3	0.087043	0.001012	0.001012	0.003014
4	0.088049	0.000999	0.001000	0.004016
5	0.087072	0.001000	0.001000	0.003001
6	0.086079	0.001000	0.000999	0.003985
7	0.089045	0.001000	0.002001	0.003001
8	0.003985	0.001011	0.001000	0.003000
9	0.089076	0.001017	0.002014	0.004014
10	0.118087	0.001011	0.001001	0.003000
11	0.087065	0.001012	0.001011	0.004017
12	0.088049	0.001000	0.001013	0.004015
13	0.087061	0.001011	0.001001	0.003002
14	0.088081	0.000996	0.001013	0.002990
15	0.089062	0.001012	0.002001	0.003013
16	0.088048	0.000999	0.001000	0.003015
17	0.088061	0.001000	0.001012	0.003989
18	0.089074	0.001000	0.001987	0.003006
19	0.087073	0.002001	0.001012	0.003001
20	0.088063	0.001000	0.001986	0.003001
21	0.088062	0.001012	0.001001	0.003002
22	0.088064	0.001000	0.000999	0.002988
23	0.086071	0.000996	0.001000	0.004017
24	0.087075	0.001000	0.000988	0.003013
25	0.087061	0.000987	0.001012	0.003010
26	0.087047	0.001000	0.001988	0.004004
27	0.089050	0.001001	0.001003	0.003014
28	0.089077	0.002000	0.001983	0.004014
29	0.089062	0.001000	0.001001	0.002988
30	0.089066	0.000999	0.001002	0.003013
Promedio:	0.086095	0.001069	0.001269	0.003305

En base a los resultados y promedios de la tabla , podemos concluir que el mejor método de ordenamiento es :

Quicksort