TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN.

SEMESTRE AGOSTO-DICIEMBRE 2018

INGENIERÍA EN SISTEMAS COMPUTACIONALES

MATERIA: Estructura de Datos
MATERIA: Ray Parra

Proyecto Final

Metodos de Ordenamiento

ALUMNO: GASTELUM LEYVA JUAN LUIS #17212135

FECHA DE ENTREGA: 30 DE NOVIEMBRE DEL 2018

Este proyecto consiste en implementar y comparar los diferentes metodos de ordenamiento (BubbleSort, QuickSort, ShellSort, MergeSort) en el código de programación Python, la duración de tiempo que toma cada método difiere dependiendo de la longitud del arreglo debido a su complejidad, unos pueden ser muy lentos con listas pequeñas y rápidos con listas grandes y viceversa. En este proyecto se tomará una lista de 10 mil elementos (1 al 10000), cada método se correrá 30 veces para tomar el tiempo de acomodo de cada vez y sacar su promedio.

Para medir la duración de tiempo del código utilizaremos la librería datetime, luego tomaremos el tiempo antes de correr el metodo de ordenamiento para la lista (now), y luego tomaremos el tiempo después de correr el método (later) utilizando la misma función, restamos el tiempo inicial al tiempo final y tendremos la diferencia que es igual a la duración de tiempo de duración del método

```
from datetime import datetime

now1 = datetime.now()

Metodo(lista)

later1 = datetime.now()
from datetime import datetime

difference1 = later1 - now1
```

Luego imprimimos la lista sin acomodar, la lista acomodada y los tiempos de duracion de todos los metodos.

```
print("Lista Original: ")
print(my_list)
print("Lista Ordenada: ")
print(list_1)
print("Duracion")
print("Bubble: ", difference1)
print("QuickSort: ",difference2)
print("ShellSort: ", difference3)
print("MergeSort: ",difference4)
```

Ahora veremos los métodos de ordenamiento y su implementación.

BubbleSort

Este es un método muy simple de baja complejidad que compara de uno por uno los elementos de la lista de un lado hacia el otro comparando el primero con el siguiente; esto hace que este método sea rápido para listas cortas pero muy lento para listas grandes.

```
def bubble(bad_list):
    length = len(bad_list) - 1
    sorted = False

    while not sorted:
        sorted = True
        for i in range(length):
            if bad_list[i] > bad_list[i+1]:
                sorted = False
                bad_list[i], bad_list[i+1] = bad_list[i+1], bad_list[i]
```

QuickSort

El método quicksort está diseñado para listas grandes de elementos, este toma un pivote en la lista y deja los mayores de un lado y los menores del otro lado del pivote, creando así 2 listas que luego une, para esto utiliza 2 funciones una que compara los elementos y los divide y otra que acomoda al pivote cuando se termina de acomodar los menores y los mayores.

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:

        splitpoint = partition(alist,first,last)

        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)

def partition(alist,first,last):
    pivotvalue = alist[first]

    leftmark = first+1
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp
    return rightmark
```

ShellSort

Este método utiliza una segmentación entre los datos. Funciona comparando elementos que estén distantes; la distancia entre comparaciones decrece conforme el algoritmo se ejecuta hasta la última fase, en la cual se comparan los elementos adyacentes, por esta razón se le llama ordenación por disminución de incrementos.

```
def ShellSort(lst):
    gap=len(lst)//2
    while gap>0:
        for i in range(0, len(lst)-gap):
            if lst[i]>lst[i+gap]:
                lst[i], lst[i+gap] = lst[i+gap], lst[i]
                for j in range(i-gap,-1,-gap):
                    if lst[j]>lst[j+gap]:
                        lst[j], lst[j+gap] = lst[j+gap], lst[j]
        gap//=2
```

MergeSort

Este método como el método quicksort aplica la técnica divide y vencerás, dividiendo la secuencia de datos en dos subsecuencias hasta que las subsecuencias tengan un único elemento, luego se ordenan mezclando dos subsecuencias ordenadas en una secuencia ordenada, en forma sucesiva hasta obtener una secuencia única ya ordenada. Si n = 1 sólo hay un elemento por ordenar, sino se hace una ordenación de mezcla de la primera mitad del arreglo con la segunda mitad. Las dos mitades se ordenan de igual forma. Siempre se lleva a cabo una última pasada.

```
def MergeSort(lst):
    half = len(lst)//2
    left_half, right_half = lst[:half], lst[half:]
    if len(left_half) > 1: left_half = MergeSort(left_half)
    if len(right_half)> 1: right_half= MergeSort(right_half)
    sorted_list = []
    while left_half and right_half:
        if left_half[0] <= right_half[0]:
            sorted_list.append(left_half.pop(0))
        else:
            sorted_list.append(right_half.pop(0))
    return sorted_list + (left_half or right_half)
```

Para respaldar las afirmaciones hechas en este proyecto se ejecutaron 30 veces cada método lo que dio como resultado la siguiente tabla:

Metodo				
Run	Bubble	QuickSort	ShellSort	MergeSort
1	13.658262	0.0244	3.806873	0.040016
2	13.353839	0.023451	3.560685	0.038038
3	14.056406	0.023424	3.66065	0.040031
4	13.497062	0.024407	3.589634	0.039068
5	13.126315	0.023421	3.564716	0.038064
6	13.01937	0.022476	3.545951	0.0375
7	13.129662	0.022449	3.74111	0.039012
8	13.347572	0.023424	3.669306	0.037088
9	13.757471	0.025377	3.834663	0.040015
10	13.881575	0.0244	3.74413	0.037682
11	13.504325	0.023424	3.605285	0.038064
12	13.085391	0.0244	3.648767	0.038091
13	13.067681	0.023424	3.577133	0.03804
14	13.234609	0.023424	3.576381	0.038063
15	13.09841	0.02633	3.603273	0.039041
16	13.185624	0.024375	3.625309	0.042944
17	13.43453	0.023424	3.56026	0.037116
18	13.440267	0.026327	3.57364	0.037082
19	12.930293	0.024905	3.579401	0.038064
20	13.415125	0.023425	3.568256	0.037087
21	13.277957	0.023455	3.556158	0.037086
22	13.863018	0.023424	3.703931	0.039985
23	13.545191	0.024399	3.744915	0.040016
24	13.726455	0.026344	3.717451	0.038064
25	12.972911	0.023424	3.535956	0.037088
26	13.195702	0.023425	3.62875	0.038064
27	13.708616	0.025349	3.7794	0.042945
28	13.724417	0.026324	3.763038	0.039544
29	13.306988	0.027328	3.751596	0.044895
30	13.695894	0.024372	3.774924	0.037094
Promedio	13.40803127	0.02428436667	3.6530514	0.03882956667

Con respecto a la lista de 10k elementos el método de ordenamiento más rápido fue el QuickSort seguido por el MergeSort, estos tuvieron casi el mismo tiempo; después esta el Shellsort con una duración pasable y por ultimo esta el Bubble que tardo demasiado.

QuickSort	0.02428436667
MergeSort	0.03882956667
ShellSort	3.6530514
Bubble	13.40803127

Conclusion

Cada herramienta tiene diferentes desempeños dependiendo para que se creó, y los métodos de ordenamiento no son diferentes, dependiendo de la lista que quieras ordenar o de los datos que quieras procesar el rendimiento puede variar, por eso es nuestro trabajo entender el funcionamiento de estos y aplicar el más óptimo para dado problema llevando a cabo nuestro trabajo como programadores qué es resolver problemas computacionales con el menor esfuerzo posible en el menor tiempo desde el punto de vista computacional.