Reporte de Algoritmo de ordenamiento

Ordenamiento de Burbuja:

Es un sencillo algoritmo de ordenamiento, pero puede ser de los más tardados en el peor de los casos. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Su nombre es debido de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo.

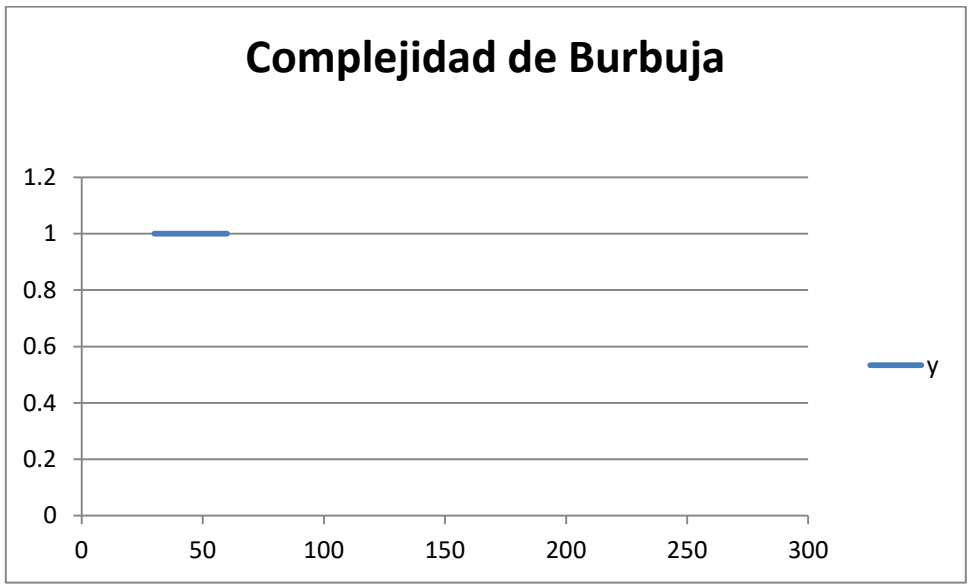
El procedimiento de la burbuja es el siguiente: § Ir comparando desde la casilla 0 número tras número hasta encontrar uno mayor, si este es realmente el mayor de todo el vector se llevará hasta la última casilla, si no es así, será reemplazado por uno mayor que él. § Este procedimiento seguirá así hasta que haya ordenado todas las casillas del vector. § Una de las deficiencias del algoritmo es que ya cuando ha ordenado parte del vector vuelve a compararlo cuando esto ya no es necesario.

Su orden O(n2) lo hace muy ineficiente para usar en listas que tengan más que un número reducido de elementos. Incluso entre los algoritmos de ordenamiento de orden O(n2), otros procedimientos como el ordenamiento por inserción son considerados más eficientes. Dada su simplicidad, el ordenamiento de burbuja es utilizado para introducir el concepto de algoritmo de ordenamiento para estudiantes de ciencias de la computación. A pesar de esto, algunos investigadores como Owen Astrachan han criticado su popularidad en la enseñanza de ciencias de la computación, llegando a recomendar su eliminación de los planes de estudio.

```
#Declaracion de funciones

def burbuja(a):

    for i in range (0, len(A)):

        for j in range(0, len(A)-1):

            if(A[J+1]<A[j]:

                aux=A[j]

                A[j]=A[j+1]

                A[j+1]=aux

        print(A)

#Programa Principal

A=[6,5,3,1,8,7,2,4]
```

print(A)

burbuja(A)



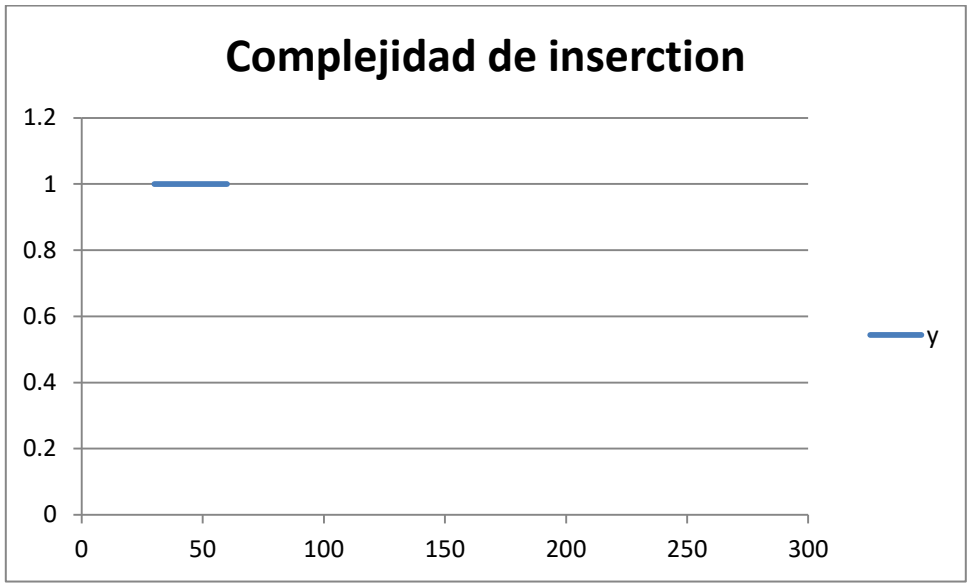
Ordenamiento por inserción

Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria. Requiere $O(n^2)$ operaciones para ordenar una lista de n elementos. Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

cnt = 0

```
def orden.por.inserccion(array):
        global cnt
        for indice in range (1,len(array)):
                valor=array[indice]
                i=indice-1
         while i>=0:
```

```
                cnt+=1
            if valor<lista[i]:
            array[i+1]=array[i]
                        array[i]=valor
            i-=1
        else:
                break

    return array
```



Ordenamiento por selección

El ordenamiento por selección es un algoritmo de ordenamiento que requiere operaciones para ordenar una lista de n elementos.

Su funcionamiento es el siguiente:

 Buscar el mínimo elemento de la lista Intercambiarlo con el primero Buscar el siguiente mínimo en el resto de la lista Intercambiarlo con el segundo

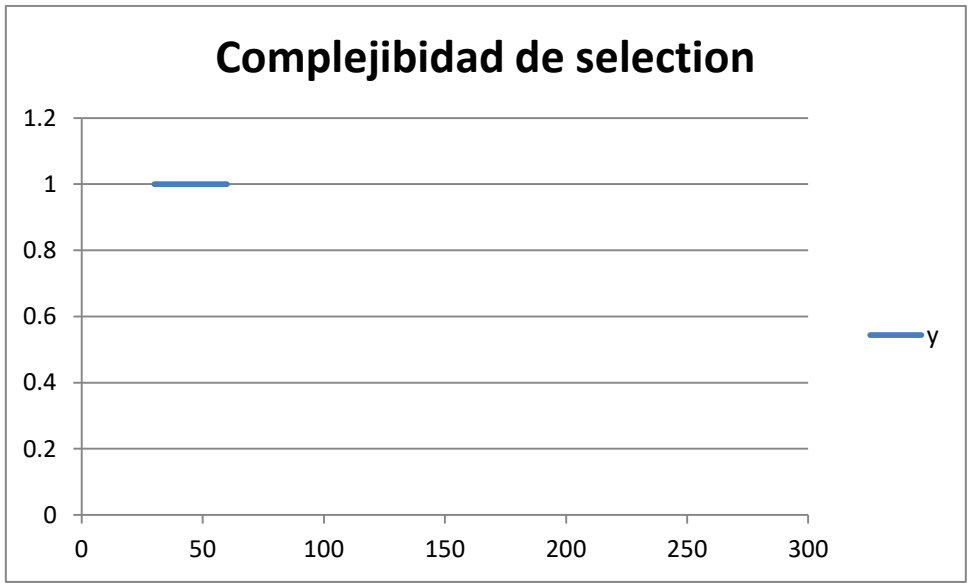
Y en general:

 Buscar el mínimo elemento entre una posición i y el final de la lista Intercambiar el mínimo con el elemento de la posición i

Al algoritmo de ordenamiento por selección, para ordenar un vector de **n** términos, tiene que realizar siempre el mismo número de comparaciones:

$C(n)= n^2 –n/2$

```
def selection(arr):
    for i in range(0,len(arr)-1):
        val = 1
        for j in range(i+1,len(arr)):
            if arr[j]< arr[val]:
                val = j
        if val != i:
            aux = arr[i]
            arr[i]=arr [val]
            arr [val]=aux
    return arr


A=[5,4,7]
print(A)
selection(a)
print(a)
```



Ordenamiento de Quicksort

El algoritmo trabaja de la siguiente forma:

 Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote. Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha. Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

 En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $O(n \cdot \log n)$. En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende

del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente. En el caso promedio, el orden es $O(n \cdot \log n)$.

import random

```
cnt=0


def quicksort(a):
    global cnt
    if len(a)<2:
        return a
    p=a.pop(0)
    menores, mayores = [], []
    for e in a:
        cnt+=1
        if e<= p:
            menores.append(e)
        else:
            mayores.append(e)
    return quicksort(menores)+[p]+ quicksort(mayores)




#programa principal
a=[4,5,78,2,8]
print(a)
quicksort(a)
print(a)
```

```
print(cnt)
```

Conclusion

Durante este reporte estuvimos viendo algunos algoritmos en si burbuja, inserction, selection y quicksort y como podemos notar en las graficas el algortimo de burbuja e inserction representa casi lo mismo se va incrementando de igual manera mientras que selection tiende a ir mas para arriba los valores y el de quicksort aparte de que es mas rápido es mas útil para estas operaciones