Reporte de algoritmos de ordenamiento

Haydee Judith Arriaga Ponce

Matricula: 1659539

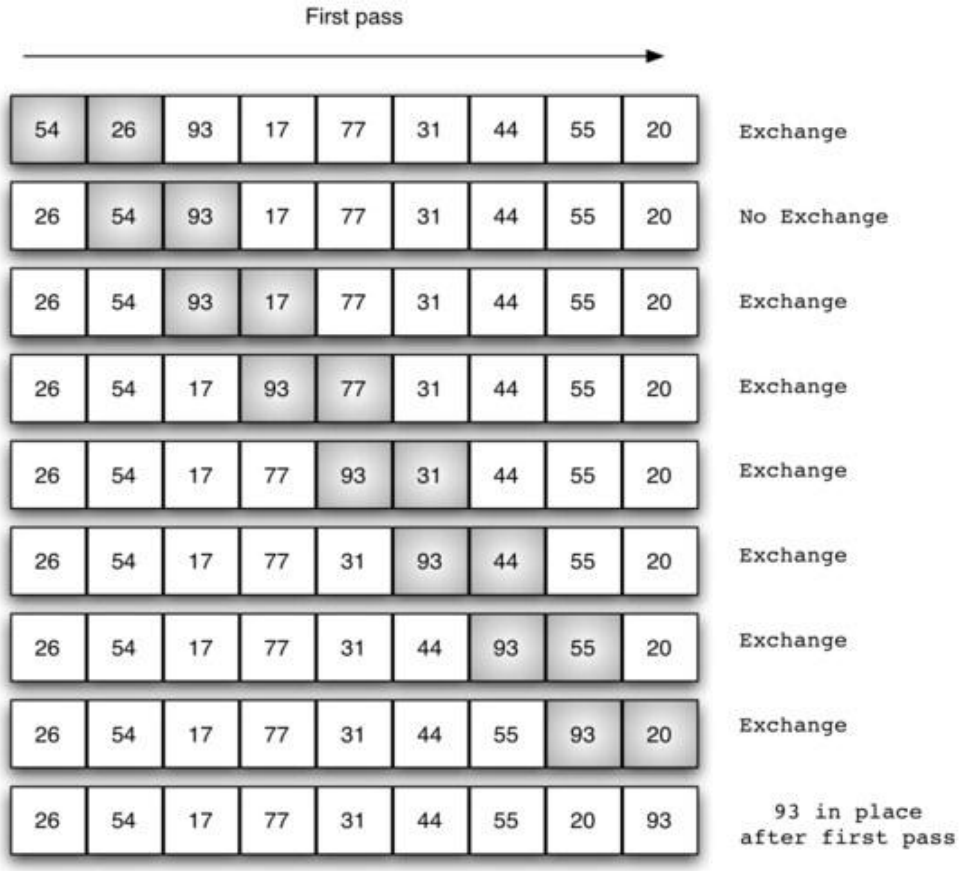
l.judithmat@gmail.com

https://github.com/haydee10arriaga/1659539_Mat.Com

1 de septiembre de 2017

1. BUBBLE SORT

Hace múltiples pases a través de una lista. Compara los artículos adyacentes e intercambia los que están fuera de servicio. Cada paso a través de la lista coloca el siguiente valor más grande en su lugar apropiado. En esencia, cada elemento "burbujea" hasta el lugar al que pertenece.



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 esta ordenado parte del vector vuelve a compararlo cuando esto ya no es necesario.

Al algoritmo de la burbuja, para ordenar un vector de n términos, tiene que realizar siempre el mismo número de comparaciones. Esto es, el número de comparaciones c(n) no depende del orden de los términos, si no del número de términos.

$$\Theta(c(n)) = n^2.$$

La complejidad es n*n = O(n^2).

BUBBLE SORT ALGORITMO

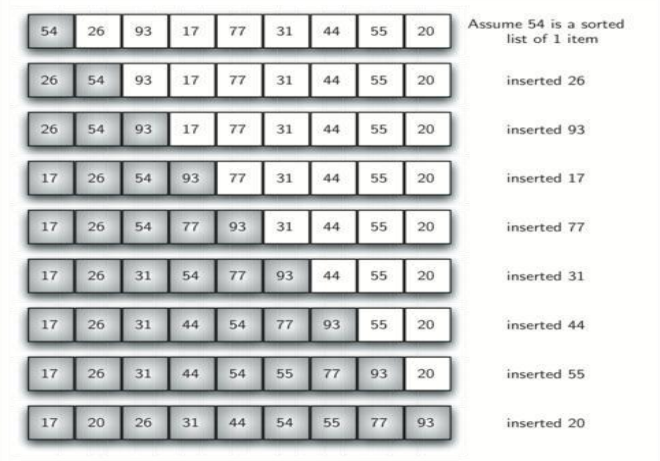
```
#Haydee Judith Arriaga Ponce
#bubble algoritmo de ordenacion

def burbuja(A):
        cnt=0
        for i in range(1,len(A)):
                        for j in range(0, len(A)-1):
                                cnt+=1
                                if(A[j+1]<A[j]):
                                        aux=A[j]
                                        A[j]=A[j+1]
                                        A[j+1]=aux

        return cnt
```

2. INSERTION SORT

El algoritmo por inserción (Insertion Sort) consiste en insertar cada elemento (partiendo por el segundo) en la posición correcta, desplazando cada elemento que sea mayor a su izquierda, hasta encontrar un elemento que sea menor, insertando el elemento a la derecha de este.



El tipo de inserción es un algoritmo de clasificación simple que funciona de la misma manera que ordenamos las cartas en nuestras manos.

Tiempo Complejidad: O (n * n)

Espacio Auxiliar: O (1)

Aquí está otra forma de pensar acerca del ordenamiento. Imagina que estás jugando un juego de cartas. Tienes las cartas en tu mano y las cartas están ordenadas. Tomas exactamente una nueva carta del mazo. La tienes que colocar en el sitio correcto de manera que las cartas en tu mano sigan estando ordenadas. En el ordenamiento por selección, cada elemento que agregas al subarreglo ordenado es mayor o igual que los elementos que ya están en el subarreglo ordenado. Pero en nuestro ejemplo de las cartas, la nueva carta podría ser menor que algunas de las cartas que ya tienes en la mano, así que vas una por una, comparando la nueva carta con cada una de las que ya tienes en la mano, hasta encontrar el lugar donde debe ser colocada. Insertas la nueva carta en el sitio correcto y, una vez más, tienes en la mano cartas completamente ordenadas. Entonces tomas otra carta del mazo y repites el mismo procedimiento. Luego otra carta, y otra, y así sucesivamente, hasta terminar con el mazo.

Esta es la idea detrás del ordenamiento por inserción. Itera sobre las posiciones en el arreglo, comenzando con el índice 1. Cada nueva posición es como la nueva carta que tomas del mazo, y necesitas insertarla en el sitio correcto en el subarreglo ordenado a la izquierda de esa posición.

Casos de límite: El tipo de inserción toma un tiempo máximo para ordenar si los elementos se ordenan en orden inverso. Y toma tiempo mínimo (Orden de n) cuando los elementos ya están ordenados.

Usos: El tipo de inserción se utiliza cuando el número de elementos es pequeño. También puede ser útil cuando la matriz de entrada está casi ordenada, sólo unos pocos elementos están fuera de lugar en una matriz completa.

INSERTION SORT ALGORITMO

```
def orden_por_insercion(array):
    cnt=0
    for indice in range(1,len(array)):
        i = indice
        #valor=array[indice] #valor es el elemento que vamos a comparar
        #i=indice-1 #i es el valor anterior al elemento que estamos comparando
        while i>0 and array[i]<array[i-1]:
            cnt+=1
            aux = array[i-1]
            array[i-1]=array[i] #intercambiamos los valores
            array[i]=aux
            i-=1 #decrementamos en 1 el valor de i
    return cnt
```

3. SELECTION SORT

El tipo de selección mejora en el tipo de burbuja haciendo sólo un intercambio por cada paso a través de la lista. Este algoritmo mejora ligeramente el algoritmo de la burbuja. En el caso de tener que ordenar un vector de enteros, esta mejora no es muy sustancial, pero cuando hay que ordenar un vector de estructuras más complejas, la operación de intercambiar los elementos sería más costosa en este caso. Su funcionamiento se puede definir de forma general como:

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

Intercambiar el mínimo con el elemento de la posición i

Así, se puede escribir el siguiente pseudocódigo para ordenar una lista de n elementos indexados desde el 1:

para i=1 hasta n-1;

Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenarlo todo. Su implementación requiere O(n2) comparaciones e intercambios para ordenar una secuencia de elementos.

SELECTION SORT ALGORITMO

```
#Haydee Judith Arriaga Ponce
#algoritmo de ordenacion selection

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

4. QUICKSORT

Es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n.

Una orden rápida selecciona primero un valor, que se llama el valor de pivote. Aunque hay muchas formas diferentes de elegir el valor de pivote, simplemente usaremos el primer elemento de la lista. El papel del valor de pivote es ayudar a dividir la lista. La posición real en la que pertenece el valor pivote en la lista ordenada final, comúnmente denominada punto de división, se utilizará para dividir la lista para las llamadas posteriores a la ordenación rápida.

El algoritmo consta de los siguientes pasos:

- 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.

Eficiencia del algoritmo

La eficiencia del algoritmo depende de la posición en la que termine el pivote elegido. 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 $O(n \bullet \log n)$. 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•log n). Y no es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

Funcionamiento

Se debe llamar a la función Quicksort desde donde quiera ejecutarse.

Ésta llamará a colocar pivote para encontrar el valor del mismo.

Se ejecutará el algoritmo Quicksort de forma recursiva a ambos lados del pivote.

QUICK SORT ALGORITMO

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

CONCLUSION DE LA COMPLEJIDAD DE LOS ALGORITMOS

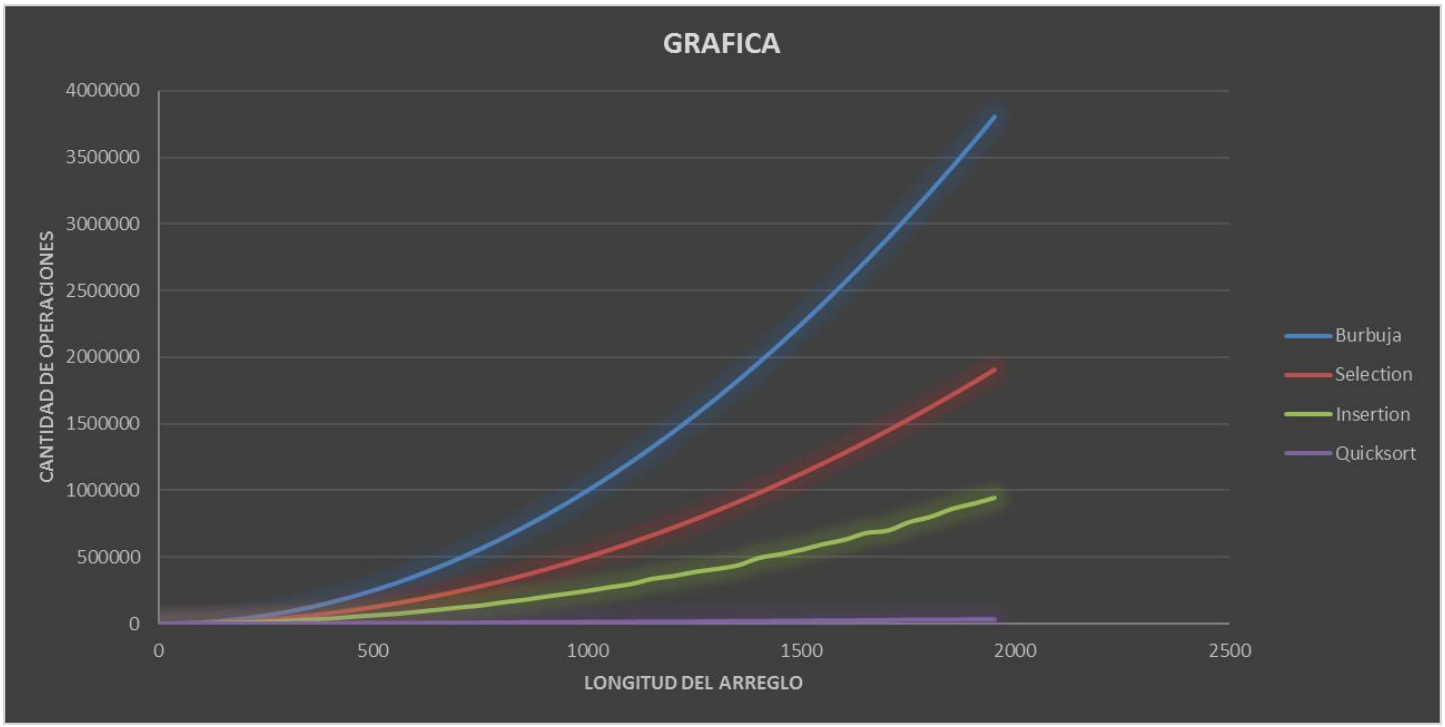
Al concluir con los 4 algoritmos que realizamos en Python Bubble, Insertion, Selection, Quicksort; con los cuales se puede probar se puede partir de un arreglo y compáralos los 4 con el mismo para visualizar la gran diferencia que hay entre ellos a simple vista el algoritmo Bubble es el que realiza más operaciones para arreglar el arreglo utilizamos una función que dada una longitud del arreglo, regrese hacia un arreglo con números aleatorios. La cual es :

```
def ran_num(n,lim_inf=0, lim_sup=100):
    arreglo = []
    for i in range(n):
        arreglo.append(random.randint(lim_inf, lim_sup))
    return arreglo
```

Con la cual se utilizó para comparar la cantidad de comparaciones u operaciones que realiza cada algoritmo. Con lo cual obtuvimos:

Longitud	Burbuja	Selection	Insertion	Quicksort
2	1	1	0	1
52	2601	1326	702	314
102	10201	5151	2511	612
152	22801	11476	6016	1106
202	40401	20301	9544	1615
252	63001	31626	15432	2104
302	90601	45451	21954	2581
352	123201	61776	31479	3281
402	160801	80601	39367	4067
452	203401	101926	52435	4489
502	251001	125751	63374	4935
552	303601	152076	74024	5726
602	361201	180901	89485	5912
652	423801	212226	105126	7728
702	491401	246051	122452	7436
752	564001	282376	137063	8662
802	641601	321201	160746	9357
852	724201	362526	178677	11433
902	811801	406351	203124	11414
952	904401	452676	224883	12059
1002	1002001	501501	247217	14188
1052	1104601	552826	274081	14176
1102	1212201	606651	297484	14787
1152	1324801	662976	337123	16722
1202	1442401	721801	357977	16469
1252	1565001	783126	389257	17412
1302	1692601	846951	411014	18834
1352	1825201	913276	438137	20100
1402	1962801	982101	494099	20748
1452	2105401	1053426	521608	22245
1502	2253001	1127251	555807	23196
1552	2405601	1203576	597110	24136
1602	2563201	1282401	630743	24896
1652	2725801	1363726	680940	26398
1702	2893401	1447551	698758	27887
1752	3066001	1533876	761924	30991
1802	3243601	1622701	801813	31216
1852	3426201	1714026	861139	31036
1902	3613801	1807851	900369	33703
1952	3806401	1904176	944523	33968

La grafica



A simple vista bubble es el que más operaciones realiza se podría decir el más tardado aunque no necesita almacenamiento temporal pero los 4 son algoritmos de ordenación útiles, prácticos y rápidos depende mucho para lo que se utilice cada uno cuenta con sus ventajas y desventajas.

En el caso de bubble e selection es muy similar su crecimiento, de los cuatro algoritmos sabemos y corroboramos que insertion es el que tiene mejor desempeño debido a que toma ventaja del orden de los números para realizar menos operaciones pero ofrece un espacio mínimo a pesar de eso sabemos que bubble, insertion y selection tienen la misma complejidad n*2 pero Quicksort tuvo una notoria disminución en la cantidad de operaciones ya que me maneja de diferente forma se tomó en él un elemento pivote donde compara y acomoda es de gran ventaja solo es cuestión de dominarlo comprenderlo al momento de acomodarlo.

En si todos los algoritmos de ordenación son una gran herramienta son útiles para nosotros solo es cuestión de comprender, analizar y así poder seguir manejándolos y desarrollando en Python.