Reporte de algoritmos de ordenamiento

Haydee Judith Arriaga Ponce

Matricula: 1659539

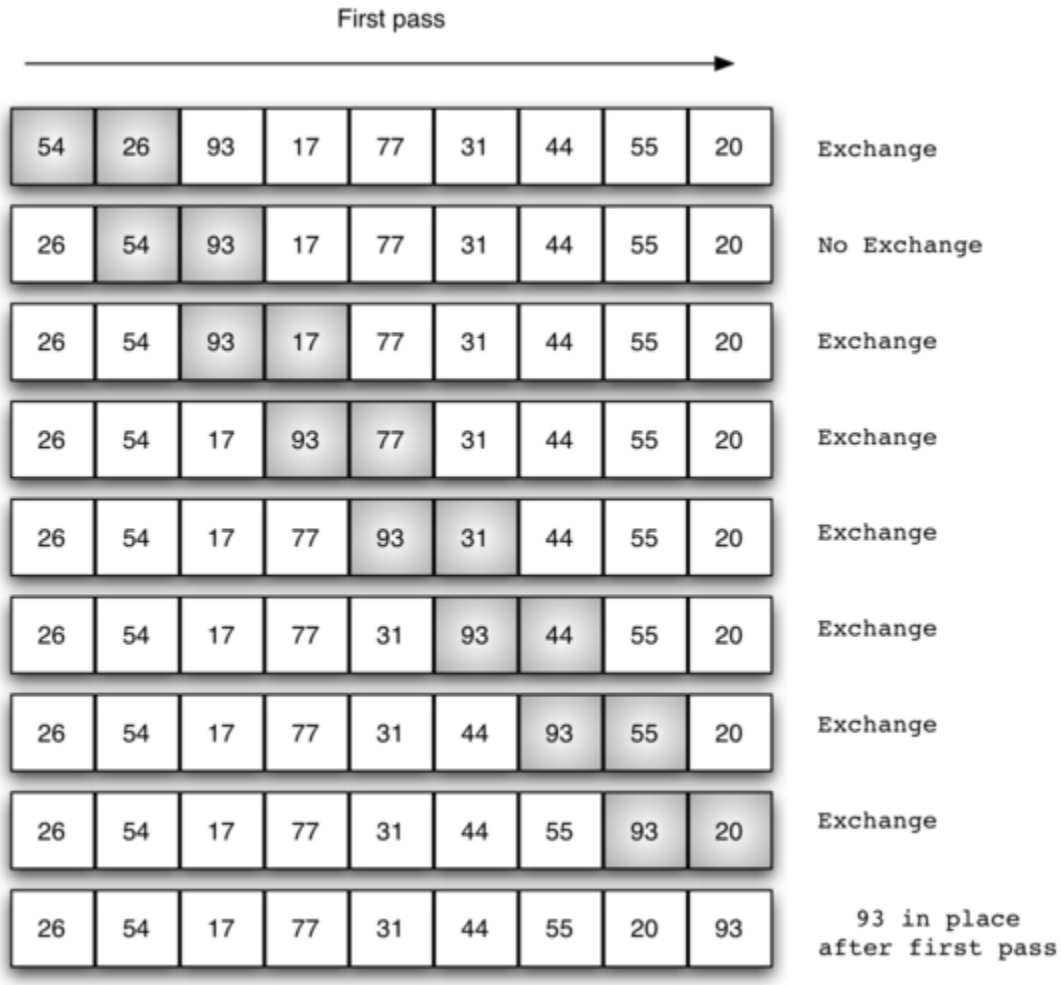
l.judithmat@gmail.com

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).

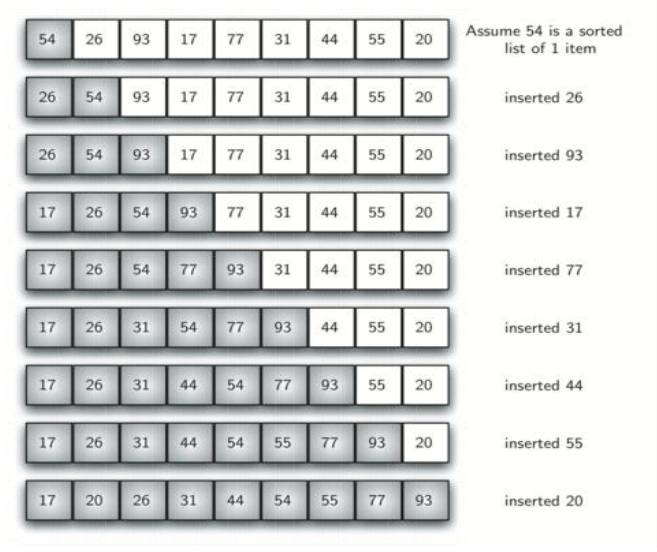
BUBBLE SORT PSEUDOCODE

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

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 PSEUDOCODIGO

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
         alist[position]=alist[position-1]
         position = position-1

     alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)

```

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

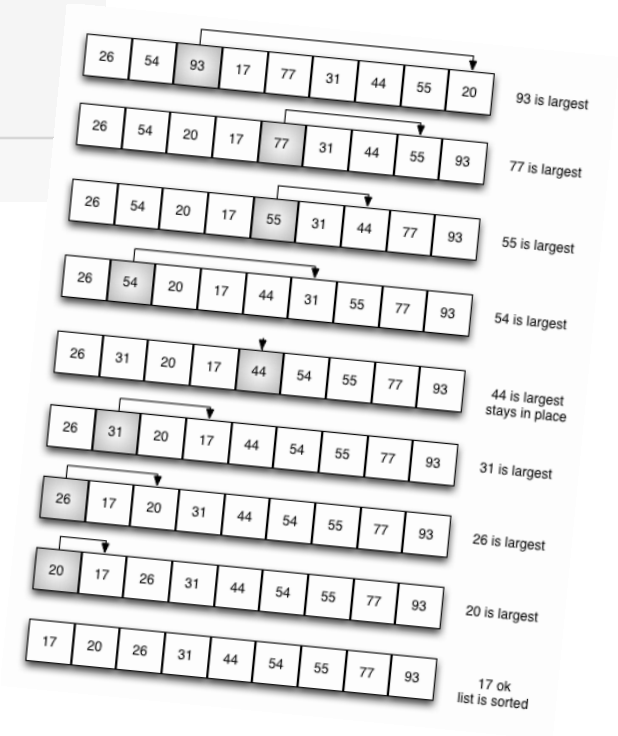
Así, se puede escribir el siguiente pseudocódigo para ordenar una lista de n elementos indexados desde el 1:

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 PSEUDOCODIGO

```
    minimo = i;
    para j=i+1 hasta n
        si lista[j] < lista[minimo] entonces
            minimo = j
        fin si
    fin para
    intercambiar(lista[i], lista[minimo])
fin para
```



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 $O(n \cdot \log n)$. 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•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 PSEUDOCODIGO

```
01:  Var
02:     m: array [1..251,1..250] of extended;
03:
04:  Procedure Gauss_Jordan;
05:            var t: extended;
06:                i,j,k,p: longint;
07:            begin
08:            for i:= 1 to n do
09:                begin
10:                for j:= i to n do
11:                    if (m[i,j]<>0) then
12:                       begin
13:                       p:= j;
14:                       break;
15:                       end;
16:                for j:= 1 to n+1 do
17:                    begin
18:                    t:= m[j,p]; m[j,p]:= m[j,i]; m[j,i]:= t;
19:                    end;
20:                for j:= 1 to n do
21:                    begin
22:                    if (j=i) then continue;
23:                    t:= -m[i,j]/m[i,i];
24:                    for k:= 1 to n+1 do
25:                        m[k,j]:= m[k,j] + t*m[k,i];
26:                    end;
27:                end;
28:            for i:= 1 to n do
29:                begin
30:                m[n+1,i]:= m[n+1,i]/m[i,i];
31:                m[i,i]:= 1;
32:                end;
33:            end;
```