Alicia Magaly Azúa Saucedo

1743217

Reporte de Algoritmos de Ordenamiento

08 abril de 2018

En este documento hablaremos de cómo trabajan los algoritmos de ordenamiento como lo son bubble, inserción, selección y Quicksort veremos cómo funcionan y sus códigos entre otras cosas.

Empezaremos con el algoritmo de **bubble:**

Este algoritmo es uno de los algoritmos menos eficientes repite varias veces la lista para ordenar, compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto. El paso a través de la lista se repite hasta que la lista este completamente ordenada. El algoritmo, que es un tipo de comparación, tiene el nombre de la forma en que los elementos más pequeños o más grandes "burbuja" a la parte superior de la lista. Aunque el algoritmo es simple, es demasiado lento e impráctico para la mayoría de los problemas.

En el peor de los casos el algoritmo de burbuja tiene una complejidad de O(n^2).

A continuación, veremos un ejemplo del algoritmo donde Tomemos el conjunto de números "5 1 4 2 8", y ordenamos el conjunto del número más bajo al número más grande usando el tipo de burbuja. En cada paso, los elementos escritos en negrita están siendo comparados.

Se necesitarán tres pasos.

Primer paso (5 1 4 2 8)(1 **5** 4 2 8), Aquí, el algoritmo compara los dos primeros elementos, y los intercambios desde 5> 1.

(1 **5 4** 2 8) (1 **4 5** 2 8), Swap desde 5> 4

(1 4 **5 2**8) (1 4 **2 5** 8), Swap desde 5> 2

(1 4 2 **5 8**) (1 4 2 **5 8**), Ahora, puesto que estos elementos ya están en orden (8> 5), el algoritmo no los intercambia.

Segundo paso

(1 **4** 2 5 8) (1 **4** 2 5 8)

(1 **4 2** 5 8) (1 **2 4** 5 8), Swap desde 4> 2

(1 2 **4 5** 8) (1 2 **4 5** 8)
(1 2 4 **5 8)** (1 2 4 **5 8)** Ahora, el conjunto ya está ordenado, pero el algoritmo no sabe si se ha completado. El algoritmo necesita un pase **entero** sin **ningún** intercambio para saber que está ordenado.

Tercer paso

(1 **2** 4 5 8) (1 **2** 4 5 8)

(1 **2 4** 5 8) (1 **2 4** 5 8)

(1 2 **4 5** 8) (1 2 **4 5** 8)

(1 2 4 **5 8)** (1 2 4 **5 8)**

Código:

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

Después continuaremos con el algoritmo de **insertion sort:**

El algoritmo de inserción es un algoritmoque,iterando la matriz, creciendo una lista ordenada detrás de ella. En cada posición de la matriz, comprueba el valor allí contra el valor más grande en la lista ordenada (que pasa a estar al lado de ella, en la posición de matriz anterior marcada). Si es más grande, deja el elemento en su lugar y se mueve a la siguiente. Si es menor, encuentra la posición correcta dentro de la lista ordenada, desplaza todos los valores mayores hasta hacer un espacio e inserta en esa posición correcta.

La entrada de peor caso más simple es una matriz ordenada en orden inverso. El conjunto de todas las entradas de peor caso consiste en todas las matrices donde cada elemento es el más pequeño o el segundo más pequeño de los elementos antes de él. En estos casos, cada iteración del bucle interno explorará y desplazará toda la subsección clasificada de la matriz antes de insertar el siguiente elemento. Esto le da a la inserción un orden cuadrático del tiempo de ejecución (es decir, O (n^ 2)).

En seguida mostrare un ejemplo de este tipo de algoritmo utilizando el siguiente conjunto (3,7,4,9,5,2,6,1) En cada paso, se subraya la clave en cuestión. La clave que se movió (o se dejó en su lugar porque era la más grande aún considerada) en el paso anterior se muestra en negrita.

3 7 4 9 5 2 6 1
37 4 9 5 2 6 1

3 **7**<u>4</u> 9 5 2 6 1
3 **4** 7 <u>9</u> 5 2 6 1
3 4 7 **9** <u>5</u> 2 6 1
3 4 **5** 7 9 <u>2</u> 6 1
2 3 4 5 7 9 <u>6</u> 1
2 3 4 5 **6** 7 9 <u>1</u>
1 2 3 4 5 6 7 9

A continuación, pondremos un código de el algoritmo de inserción:

```
def ordenamiento_insersion(A):
    global cnt
for i in range (1,len(A)):
        cnt+=1
        valor= A[i]
        q= i-1
        while q >= 0:
            if valor < A[q]:
A[q+1] = A[q]
                A[q] = valor
                q = q-1
            else:
                break
```

Otro de los algoritmos de ordenamiento es el algoritmo **selección**:

Este tipo de algoritmo de ordenamiento busca el numero mas pequeño en la lista y lo intercambia en la primera posición luego busca el otro más pequeño de la lista y lo intercambia en la segunda posición y así se va con todos los números de la lista hasta que la lista ya está ordenada. En el peor de los casos el algoritmo tiene una complejidad de O(n^2)

Un ejemplo de este tipo de algoritmo de ordenamiento utilizando la lista 3,8,9,1,5 donde el algoritmo ira buscando el más pequeño donde los subrayados son los números que se van a iterar uno es el máspequeño y el de la posición un y el de negritas son los numero ya iterados

<u>3</u> 8 9 <u>1</u> 5

1 8 9 **3** 5

1 <u>8</u> 9 <u>3</u> 5

1 **3** 9 **8** 5

1 3 <u>9</u> 8 <u>5</u>

1 3 **5** 8 **9**

1 3 5 8 9

1 3 5 8 9

Y así se muestra que la lista ya esta ordenada

A continuación, mostraremos un código de este tipo de algoritmo de ordenamiento:

1. **PROCEDIMIENTO** selection_sort (Vector a[1:n])
2. **PARA** i **VARIANDO DE** 1 **HASTA** n - 1 **HACER**
3. ENCONTRAR [j] EL ELEMENTO MÁS PEQUEÑO DE [i + 1:n];
4. INTERCAMBIAR [j] Y [i];
5. **FIN PROCEDIMIENTO**;

otro de los algoritmos de ordenamiento de los cuales hablaremos es de **QuickSort:**

El algoritmo trabaja eligiendo un elemento de la lista de elementos a ordenar, al que llamamos **pivote**. Coloca 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·log n)**.En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $\mathbf{O(n^2)}$.

Un código para este algoritmo de ordenamiento es:

```
def ordenamiento_quicksort(A):

    global qc

if len(A) < 2:

        return A

    p= A.pop(0)

    menores, mayores= list(), list()

    for c in A:
```

```
        qc+=1
        if c <=p:
            menores.append(c)
        elif c > p:
            mayores.append(c)
    return ordenamiento_quicksort(menores) + [p] + ordenamiento_quicksort(mayores)
```