Reporte

Algoritmos de ordenación

Karen Eunice Oviedo Garza 1753764

En este reporte hablaremos sobre los 4 algoritmos de ordenación: insertion, bubble, selection, quicksort.

Insertion

Inicialmente se tiene un solo elemento en un conjunto ordenado, cuando hay k elementos ordenados de menor a mayor se toma el elemento k+1 y se compara con todos los elementos ya ordenados

1. Se parte de una lista

Lista = | 3 | 2 | -1 | 5 | 0 | 2 |

2. Se considera el segundo elemento de la lista, y ordenarlo respecto del primero, desplazándolo hasta la posición correcta, si corresponde.

Lista = | 2 | 3 | -1 | 5 | 0 | 2 |

Se desplaza el valor 2 antes de 3.

3. Se considera el tercer elemento de la lista, y ordenarlo respecto del primero y el segundo, desplazándolo hasta la posición correcta, si corresponde.

Lista = | -1 | 2 | 3 | 5 | 0 | 2 |

Se desplaza el valor —1 antes de 2 y de 3.

4. Se considera el cuarto elemento de la lista, y ordenarlo respecto del primero, el segundo y el tercero, desplazándolo hasta la posición correcta, si corresponde.

El 5 está correctamente ubicado respecto de —1, 2 y 3 (como el segmento hasta la tercera posición está ordenado, basta con comparar con el tercer elemento del segmento para verificarlo).

N-1. Todos los elementos excepto el ante-último ya se encuentran ordenados.

N. Se considera el N–ésimo elemento de la lista, y ordenarlo respecto del segmento formado por el primero hasta el N − 1–ésimo, desplazándolo hasta la posición correcta, si corresponde.

Lo peor que le puede pasar a un elemento que está en la posición j es que deba ser ubicado al principio de la lista. Y lo peor que le puede pasar a una lista es que todos sus elementos deban ser reubicados.

```
def ord_insercion(lista):
    for indice in range(1,len(lista)):
        valor=lista[indice]
        i=indice-1
        while i>=0:
            if valor<lista[i]
                lista[i+1]=lista[i]
                lista[i]=valor
                i=i-1
            else:
                break
```

Requiere **O(n^2)** operaciones para ordenar una lista de **n** elementos.

Bubble

Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

1. Se comparan los dos primeros elementos, si el segundo es superior al primero, se dejan tal como están, pero si el primero es el más grande, se intercambian los elementos.

2. A continuación se compara el segundo elemento con el tercero aplicando los mismos criterios del paso anterior.

3. De esta forma se repite la operación de comparación con todos los elementos que forman el arreglo. Cuando se alcance el último elemento se ha encontrado el elemento que tiene el valor más elevado y este ha quedado situado al final del vector.

4. Para encontrar el segundo elemento se repite la operación de ordenación de todos los elementos del arreglo excepto con el último.

Reiterando el proceso descrito, el arreglo quedará ordenado cuando solo se comparen los dos primeros elementos.

```
elementos = [1,3,5,4,7,9,8,6]
numero = len(elementos)
```

```
i= 0 while (i <
numero):
   j = i     while (j < numero):
if(elementos[i] > elementos[j]):
temp = elementos[i]
elementos[i] = elementos[j]
        elementos[j] = temp
j= j+1
   i=i+1

for elemento in elementos:
   print elemento
```

```
def burbuja(A):
      for i in range(1,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)
```

Tiene una complejidad de $O(n^2)$

Selection

-No requiere memoria adicional -Es
lento

1. Buscar el mayor de todos los elementos de la lista.

Lista = | 3 | 2 | -1 | 5 | 0 | 2 |

Encuentra el valor 5 en la posición 3.

2. Poner el mayor al final (intercambiar el que está en la última posición de la lista con el mayor encontrado).

Intercambia el elemento de la posición 3 con el de la posición 5. En la última posición de la lista está el mayor de todos.

Lista = | 3 | 2 | -1 | 2 | 0 | 5 |

3. Buscar el mayor de todos los elementos del segmento de la lista entre la primera y la anteúltima posición.

Lista = | 3 | 2 | -1 | 2 | 0 | 5 |

Encuentra el valor 3 en la posición 0.

4. Poner el mayor al final del segmento (intercambiar el que está en la última posición del segmento, o sea anteúltima posición de la lista, con el mayor encontrado).

Lista = | 0 | 2 | -1 | 2 | 3 | 5 |

Intercambia el elemento de la posición `0` con el valor de la posición `4`. En la anteúltima y última posición de la lista están los dos mayores en su posición definitiva.

N: Se termina cuando queda un único elemento sin tratar: el que está en la primera posición de la lista, y que es el menor de todos porque todos los mayores fueron reubicados.

Lista = | -1 | 0 | 2 | 2 | 3 | 5 |

La función principal es recorrer la lista, ubicando el mayor elemento al final del segmento y luego reduciendo el segmento a analizar.

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

Tiene una complejidad de $O(n^2)$

Quicksort

1. Si lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. Se devuelve lista tal cual.

2. De lo contrario:

   - Dividir la lista en tres, usando `_partition`.

   - Llamar a `quick_sort(menores)`, `quick_sort(mayores)`, y concatenarlo con medio en el medio.

Por otro lado, en cuanto a la **función** _partition(lista):

1. Tiene como precondición que la lista es no vacía.
2. Se elige el primer elemento como pivote.
3. Se inicializan como vacías las listas menores y mayores.
4. Para cada elemento de la lista después del primero: ○ Si es menor que el pivote, se lo agrega a menores. ○ De lo contrario, se lo a agrega a mayores.

```
cnt=0
def quicksort(arr):
        global cnt
        if len(arr)<=1:
                return arr
        piv=arr[0]
        izq=[]
        der=[]
        for i in range(1,len(arr)):
                if(arr[i]<piv):
                        izq.append(arr[i])
                else:
                        der.append(arr[i])
                cnt+=1
        return quicksort(izq)+[piv]+quicksort(der)
```

Tiene una complejidad de n(log n)

Conclusiones

En conclusión se puede observar por medio del desarrollo que el algoritmo quicksort es mas eficiente.