Guía práctica de estudio 1

Algoritmos de ordenamiento parte 1

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaño

Facultad de Ingeniería

U.N.A.M.

Guía práctica de estudio 1

Estructura de datos y Algoritmos II

Algoritmos de Ordenamiento. Parte 1.

Objetivo: El estudiante identificará la estructura de los algoritmos de ordenamiento *Bubble Sort* y *Merge Sort*.

Al final de la práctica el estudiante habrá implementado los algoritmos en algún lenguaje de programación.

Antecedentes

- Análisis previo de los algoritmos en clase teórica.
- Manejo de arreglos o listas, estructuras de control y funciones en Python 3.

Introducción

Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar o buscar esa información de manera eficiente es deseable que está esté ordenada.

Ordenar un grupo de datos es arreglarlos en algún orden secuencial ya sea en forma ascendente o descendente.

Hay dos tipos de ordenamiento que se pueden realizar, el interno y el externo. El interno se lleva a cabo completamente en la memoria de acceso aleatorio de alta velocidad de la computadora es decir todos los elementos que se ordenan caben en la memoria principal de la computadora. Cuando no cabe toda la información en memoria principal es necesario ocupar memoria secundaria y se dice que se realiza un ordenamiento externo [2].

Los algoritmos de ordenamiento de información se pueden clasificar en cuatro grupos:

Algoritmos de inserción: Se considera un elemento a la vez y cada elemento es insertado en la posición apropiada con respecto a los demás elementos que ya han sido ordenados. Ejemplo de algoritmos que trabajan de esta manera es Inserción directa*, Shell Sort*, inserción binaria y *Hashing*.

Algoritmos de Intercambio. En estos algoritmos se trabaja con parejas de elementos que se van comparando y se intercambian si no están en el orden adecuado. El proceso se realiza hasta que se han revisado todos los elementos del conjunto a ordenar. Los algoritmos de *Bubble Sort* y *Quicksort* son ejemplos de algoritmos que trabajan de la forma mencionada.

Algoritmos de selección: En estos algoritmos se selecciona el elemento mínimo o el máximo de todo el conjunto a ordenar y se coloca en la posición apropiada. Esta selección se realiza con todos los elementos restantes del conjunto.

Algoritmos de enumeración: Se compara cada elemento con todos los demás y se determina cuántos son menores que él. La información del conteo para cada elemento indicará su posición de ordenamiento.

Algoritmo BubbleSort

El método de ordenación por burbuja (*BubbleSort* en inglés) es uno de los más básicos, simples y fáciles de comprender, pero también es poco eficiente.

La técnica utilizada se denomina ordenación por burbuja u ordenación por hundimiento y es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor a otro entonces se intercambia de posición. Los valores más pequeños suben o burbujean hacia la cima o parte superior de la lista, mientras que los valores mayores se hunden en la parte inferior.

Descripción Algoritmo

Para una lista a de n elementos numerados de 0 a $n-1$, el algoritmo requiere n-1 pasadas. Por cada pasada se comparan elementos adyacentes (parejas sucesivas) y se intercambian sus valores cuando el primer elemento es mayor que el segundo. Al final de cada pasada el elemento mayor se dice que ha burbujeado hasta la cima de la sub-lista actual.

Sea la Lista $a_0, a_1, a_2 \ldots . a_{n-1}$, entonces:

En la pasada 0 se comparan elementos adyacentes $(a_0, a_1), (a_1, a_2), (a_2, a_3), \ldots . . (a_{n-2}, a_{n-1})$

Se realizan $n-1$ comparaciones por cada pareja (a_i, a_{i+1}) y se intercambian si $a_{i+1} < a_i$. Al final de la pasada el elemento mayor de la lista estará situado en la posición $n-1$.

En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el segundo elemento de mayor valor el cual estará situado en la posición $n-2$.

Se realizan las siguientes pasadas de forma similar y el proceso termina con la pasada $n-1$ donde se tendrá al elemento más pequeño en la posición 0.

A continuación, se muestra un ejemplo del proceso con la lista {**9**,**21**,4,40,10,35}.

Primera Pasada
{**9**,**21**,4,40,10,35} --> {**9**,**21**,4,40,10,35} No se realiza intercambio
{9,**21**,**4**,40,10,35} --> {9,**4**,**21**,40,10,35} Intercambio entre el 21 y el 4
{9,4,**21**,**40**,10,35} --> {9,4,**21**,**40**,10,35} No se realiza intercambio
{9,4,21,**40**,**10**,35} --> {9,4,21,**10**,**40**,35} Intercambio entre el 40 y el 10
{9,4,21,10,**40**,**35**} --> {9,4,21,10,**35**,**40**} Intercambio entre el 40 y el 35
Segunda Pasada
{**9**,**4**,21,10,35,40} --> {**4**,**9**,21,10,35,40} Intercambio entre el 9 y el 4
{4,**9**,**21**,10,35,40} --> {4,**9**,**21**,10,35,40} No se realiza intercambio
{4,9,**21**,**10**,35,40} --> {4,9,**10**,**21**,35,40} Intercambio entre el 21 y el 10
{4,9,10,**21**,**35**,40} --> {4,9,10,**21**,**35**,40} No se realiza intercambio
{4,9,10,21,**35**,**40**} --> {4,9,10,21,**35**,**40**} No se realiza intercambio

Aunque la lista ya está ordenada se harían otras 3 pasadas.

Si la lista se representa con arreglos lineales un algoritmo en pseudocódigo es:

```
bubbleSort( A,n)
  Para i=1 hasta n
     Para j=0 hasta n-1
       Si (a[j]>a[j+1]) entonces
                  tmp=a[j]
                  a[j]=a[j+1]
                  a[j+1]=tmp
         Fin Si
     Fin Para
   Fin Para
Fin Bubble
```

En este pseudocódigo, las dos estructuras de repetición se realizan $n-1$ veces es decir se hacen $n-1$ pasadas y $n-1$ comparaciones en cada pasada. Por consiguiente, el número de comparaciones es $(n-1)*(n-1) = n^2-2n+1$, es decir la complejidad es $O(n^2)$.

Una posible mejora consiste en añadir una bandera que indique si se ha producido algún intercambio durante el recorrido del arreglo y en caso de que no se haya producido ninguno el arreglo se encuentra ordenado y se puede abandonar el método.

```
bubbleSort2 (a, n)
Inicio
bandera = 1;
pasada=0
Mientras pasada < n-1 y bandera es  igual a 1
  bandera = 0
     Para j = 0 hasta  j < n-pasada-1
        Si  a[j] >a[j+1] entonces
                  bandera = 1
                  tmp= a[j];
                  a[j] = a[j+1];
                  a[j+1] = tmp;
        Fin Si
      Fin Para
    pasada=pasada+1
  Fin Mientras
Fin
```

Con la mejora se tendrá que en el mejor de los casos la ordenación se hace en una sola pasada y su complejidad en $O(n)$. En el peor de los casos se requieren $(n-i-1)$ comparaciones y$(n-i-1)$ intercambios y la ordenación completa requiere de $\frac{n(n-1)}{2}$ comparaciones. Así la complejidad sigue siendo $O(n^2)$.

Merge Sort

Merge Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás o también conocido como divide y conquista. Esta es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño. Consta de tres pasos: Dividir el problema, resolver cada sub-problema y combinar las soluciones obtenidas para la solución al problema original

Las tres fases en esté algoritmo son:

Divide: Se divide una secuencia A de n elementos a ordenar en dos sub-secuencias de $n/2$ elementos.
Si la secuencia se representa por un arreglo lineal, para dividirlo en 2 se encuentra un número q entre p y r que son los extremos del arreglo o sub-arreglo. Figura 1.1. El cálculo se hace $q = \left\lfloor \frac{p+r}{2} \right\rfloor$.

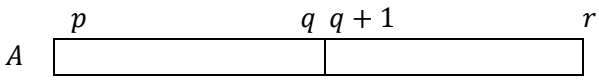

Figura 1.1.

Conquista: Ordena las dos sub-secuencias de forma recursiva utilizando el algoritmo **MergeSort**(). El caso base o término de la recursión ocurre cuando la secuencia a ser ordenada tiene solo un elemento y este se supone ordenado.

Combina: Mezcla las dos sub-secuencias ordenadas para obtener la solución.

El algoritmo en pseudocódigo queda [1]:

```
MergeSort(A,p,r)
Inicio
  Si  p<r entonces        }      // Divide
    q = ⌊(p+r)/2⌋         }
     MergeSort(A, p , q)  }      //Conquista
     MergeSort(A, q+1 , r) }
     Merge(A,p,q,r)              //  Mezcla o combina
Fin
```

La parte primordial de este algoritmo es mezclar las dos sub-secuencias de forma ordenada en la fase de **combina** y en el pseudocódigo se utiliza una función auxiliar $Merge(A, p, q, r)$ que realiza esta función. Para entender mejor como se realiza la mezcla se considera el arreglo A en la Figura 1.2.

1	2	3	4	5	6	7	8	9	10
12	9	3	7	14	11	6	2	10	5

Figura 1.2.

En un inicio se hace la llamada a la función $MergeSort(A, 1,10)$ dado que el arreglo A tiene 10 elementos y los índices de inicio y fin son 1 y 10 respectivamente. Si los índices de inicio y fin fueran 0 y 9 , entonces la llamada se haría $Merge(A, 0,9)$.

Se calcula el punto medio q que es 5 y se llama de forma recursiva a la función $MergeSort(A, 1,5)$ y $MergeSort(A, 6,10)$ sobre los subarreglos. Figura 1.3.

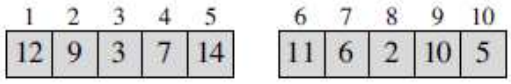

Figura 1.3.

Y después de algunas llamadas recursivas se obtienen dos sub arreglos ordenados (Figura 1.4)

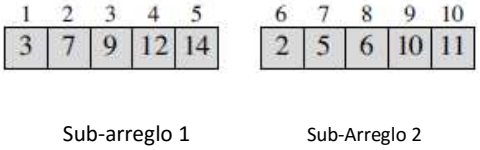

Figura 1.4.

Estos sub-arreglos se mezclan con la función $Merge(A, 1,5,10)$, tomando los valores en orden y quedando el arreglo ordenado. Así primero se escoge el 2 del sub-arreglo de la derecha luego el 3 de la izquierda, 5 y 6 de la derecha, 7 y 7 de la izquierda, 10 y 11 de la derecha y finalmente 12 y 14 de la izquierda. Figura 1.5.

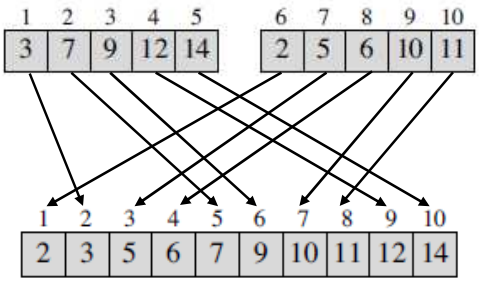

Figura 1.5.

Para el pseudocódigo de la función $Merge()$ que se muestra abajo, se considera que el índice de inicio de las secuencias es cero.

Merge(A,p,q,r)
Inicio
Formar sub-arreglo Izq[0,..., q-p] y sub-arreglo Der[0,...,r-q]
Copiar contenido de A[p...q] a Izq[0,..., q-p] y A[q+1...r] a Der[0,...,r-q -1]
i=0
j=0
Para k=p hasta r
Si (j >= r-q) ó (i < q-p+1 y Izq[i] < Der[j]) entonces
A[k]=Izq[k]
i=i+1
En otro caso
A[k]=Der[j]
j=j+1
Fin Si
Fin Para
Fin

Desarrollo

Actividad 1

Abajo se muestra la implementación en Python de los pseudocódigos de las funciones $bubbleSort()$ y $bubbleSort2()$ mencionados en la guía y que permiten realizar el ordenamiento de una lista por ***BubbleSort***. Se pide realizar un programa que ordene una lista de n elementos (la cual es proporcionada por el profesor) utilizando ambas funciones.

```
#Funcion BubbleSort
#Funcion BubbleSort Mejorada
#Autor Elba Karen Sáenz García

def bubbleSort(A):
    for i in range(1,len(A)+1):
        for j in range(len(A)-1):
            if A[j]>A[j+1]:
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp


def bubbleSort2(A):
    bandera= True
    pasada=0
    while pasada < len(A)-1 and bandera:
        bandera=False
        for j in range(len(A)-1):
            if(A[j] > A[j+1] ):
                bandera=True
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp
        pasada = pasada+1
```

Agregar la impresión para conocer el número de pasadas que realiza cada función e indicarlo. _____

¿Qué se tiene que hacer para ordenar la lista en orden inverso? _____

Actividad 2

A continuación, se proporciona la implementación en Python de los pseudocódigos de las funciones mencionadas en la guía para el algoritmo ***MergeSort***. Se requiere utilizarlas para elaborar un programa que ordene una lista proporcionada por el profesor.

Agregar en el lugar correspondiente la impresión, para visualizar las sub-secuencias divididas.

```
#MergeSort

def CrearSubArreglo(A, indIzq, indDer):
    return A[indIzq:indDer+1]

def Merge(A,p,q,r):
    Izq = CrearSubArreglo(A,p,q)
    Der = CrearSubArreglo(A,q+1,r)
    i = 0
    j = 0
    for k in range(p,r+1):
        if (j >= len(Der)) or (i < len(Izq) and Izq[i] < Der[j]):
            A[k] = Izq[i]
            i = i + 1
        else:
            A[k] = Der[j]
            j = j + 1

def MergeSort(A,p,r):
    if r - p > 0:
        q = int((p+r)/2)
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r)
```

¿Qué cambio(s) se hace(n) para ordenar la lista en orden inverso? _____

Actividad 3

Ejercicios propuestos por el profesor.

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.
Introduction to Algorithms
3rd edition MA, USA
The MIT Press, 2009

[2] KNUTH, Donald
The Art of Computer Programming
New Jersey
Addison-Wesley Professional, 2011
Volumen 3