ORDENACION

Cuestiones generales

Su finalidad es organizar ciertos datos (normalmente arrays o ficheros) en un orden creciente o decreciente mediante una regla prefijada (numérica, alfabética...). Atendiendo al tipo de elemento que se quiera ordenar puede ser:

- Ordenación interna: Los datos se encuentran en memoria (ya sean arrays, listas, etc.) y son de acceso aleatorio o directo (se puede acceder a un determinado campo sin pasar por los anteriores).

- Ordenación externa: Los datos están en un dispositivo de almacenamiento externo (ficheros) y su ordenación es más lenta que la interna.

Ordenación interna

Los métodos de ordenación interna se aplican principalmente a arrays unidimensionales. Los algoritmos de ordenación interna más simples que se estudian a continuación son muy poco eficientes para tratar gran cantidad de datos. En materias posteriores se estudiarán algoritmos más complejos que mejoran la eficiencia de la ordenación.

Selección: Este método consiste en buscar el elemento más pequeño del array y ponerlo en primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segundo lugar, y así sucesivamente hasta colocar el último elemento. Por ejemplo, si tenemos el array {40,21,4,9,10,35}, los pasos a seguir son:

```
{4,21,40,9,10,35} <-- En una primera iteración se busca el menor del
array. Se coloca el 4, el más pequeño, en primera posición: se cambia
el 4 por el 40. Luego:
{4,9,40,21,10,35} <-- Se coloca el 9, en segunda posición: se cambia
el 9 por el 21.
{4,9,10,21,40,35} <-- Se coloca el 10, en tercera posición: se cambia
el 10 por el 40.
{4,9,10,21,40,35} <-- Se coloca el 21, en tercera posición: ya está
colocado.
{4,9,10,21,35,40} <-- Se coloca el 35, en tercera posición: se cambia
el 35 por el 40.
```

```
En una primera aproximación, podemos considerar la complejidad del
algoritmo como la cantidad de instrucciones de comparación en función
de la cantidad de datos N. Así:

        -  En la 1° iteración se hacen          N – 1 comparaciones,
        -  En la 2° iteración se hacen          N – 2 comparaciones,
        -   . . .
        -  En la anteúltima iteración se hacen      2 comparaciones,
        -  En la última iteración se hacen          1 comparaciones
```

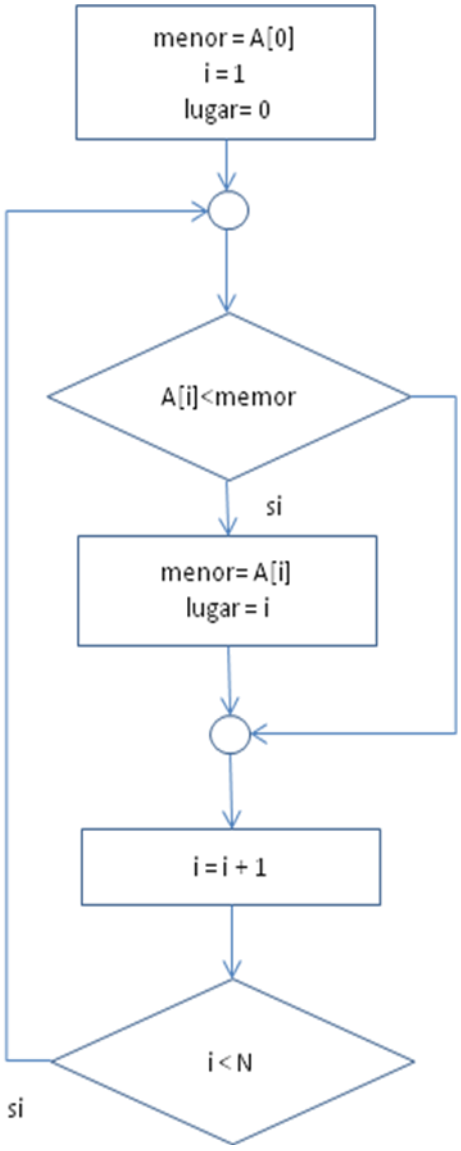
Complejidad:

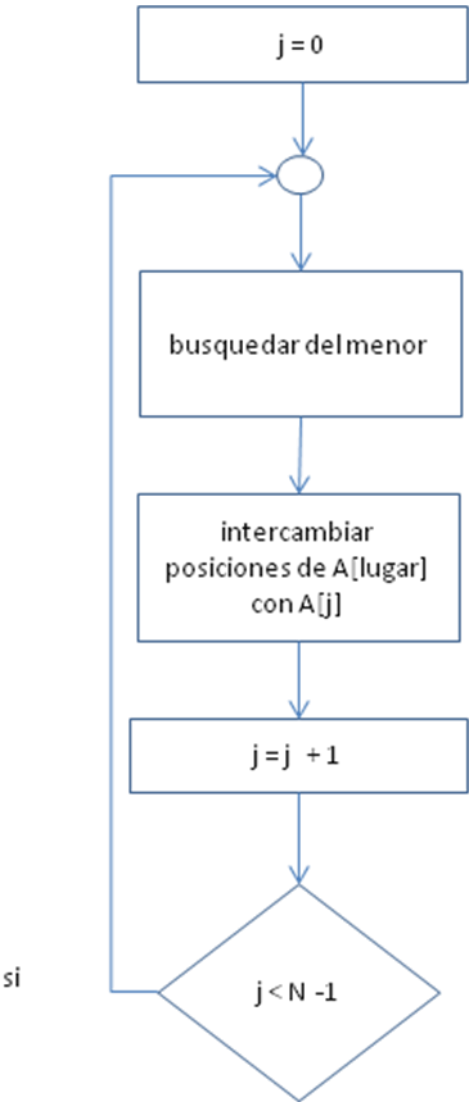
Total de comparaciones = 1 + 2 + . . . + (N - 2) + (N – 1). Se demuestra que esta serie resulta igual a $N^2/2 – N/2$, luego el tiempo de ejecución está en $O(N^2)$.

También puede buscarse el mayor y colocarlo en el último lugar, etc.:

Algoritmo de búsqueda del mayor: **Algoritmo de ordenación por selección:**

M = mayor
j = ubicación del mayor
A = arreglo
i = índice del arreglo
N = cantidad de elementos del arreglo





Algoritmo:

```
const int N=20;
int A[N];
int  aux, i, j, menor, lugar;
for (j = 0; j < N-1;j++)
{
  menor=A[j];
  lugar=j;
  for (i = j+1; i < N; i++) //busqueda del menor
  {
    if (A[i]<menor)
    {
```

```
            menor = A[i];
                    lugar = i; //posicion del menor
      };
    };
    aux = A[j];
    A[j]=A[lugar];
    A[lugar]=aux;
};
```

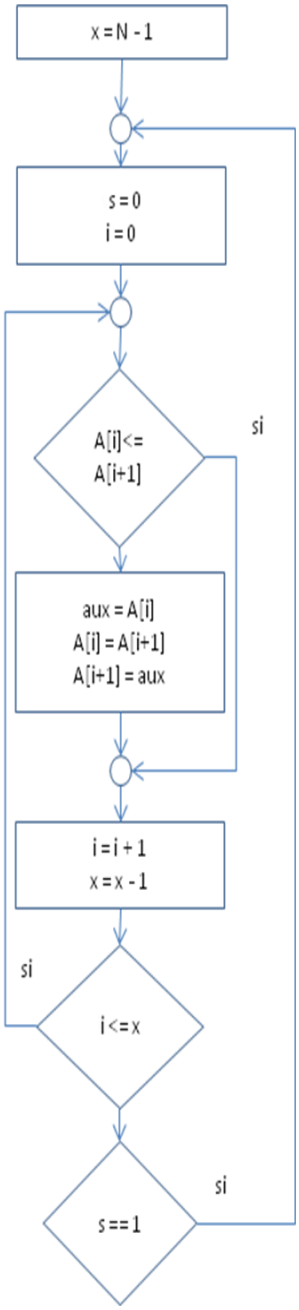
Burbujeo: Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, {40,21,4,9,10,35}:

```
Primera pasada:
{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.
{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.


En esta primera pasada, vemos que el elemento más "pesado  decantó"
quedando ordenado y los elementos "más livianos" están "burbujeando"
en ascenso a sus posiciones. La segunda pasada no necesita llegar
hasta el final porque el último ya está ordenado, así, cada pasada
reduce en 1 el número de elementos a considerar (*).

Segunda pasada:
{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.
{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.
{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.

Ya están ordenados, pero para comprobarlo habría que acabar esta
segunda pasada y hacer una tercera. Debe tenerse en cuenta que si al
final de una pasada no hay ningún intercambio significa que el array
ya está ordenado (**).
```

Complejidad:

Si el array tiene N elementos, en el peor caso, hay que hacer N-1 pasadas, y habría que hacer (N-i-1) comparaciones, para cada pasada i, es decir (N-1)* (N-i-1) comparaciones en total. El número de comparaciones es, por tanto, N(N-1)/2, lo que nos deja un tiempo de ejecución, al igual que en la selección, en $O(N^2)$. El peor caso se dá cuando el array está invertido, es decir ordenado en orden descendente. La complejidad será menor si el array está, inicialmente, parcialmente ordenado.

A = arreglo
s = bandera que indica si hubo algún cambio
i = índice del arreglo
N = cantidad de elementos del
arreglo
x = límite superior del subarreglo a ordenar



Algoritmo:

```
const int N = 20;
int A[N];
int i, j, aux;
for (i= 0; i < N-1; i++)
{
  for (j= 0; j < N-i; j++)
  {
   if (A [j+1]<A[j])
     {
      aux = A[j+1];
      A[j+1]=A[j];
      A[j]=aux;
     };
   };
};
```

```
Este algoritmo puede mejorarse tomando en cuenta las consideraciones
(*) y (**) como se indica en el diagrama de flujo.
```

Inserción directa: En este método lo que se hace es tener una sublista ordenada de elementos del array e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada. Para el ejemplo {40,21,4,9,10,35}, se tiene:

```
{40,21,4,9,10,35} <-- La primera sublista ordenada es {40}.

Insertamos el 21:
{21,40,4,9,10,35} <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:
{4,21,40,9,10,35} <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:
{4,9,21,40,10,35} <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:
{4,9,10,21,40,35} <-- Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:
{4,9,10,21,35,40} <-- El array está ordenado.
```

Complejidad:

En el peor de los casos, el número de comparaciones que hay que realizar es de N*(N+1)/2-1, lo que nos deja un tiempo de ejecución en $O(N^2)$. En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(N).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Vemos que cuanto más ordenada esté inicialmente más se acerca a O(N) y cuanto más desordenada, más se acerca a $O(N^2)$.

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

Algoritmo

```
const int N=20;
int A[N];
int i, j, aux;
               for (i=1; i<N; i++)
               {
                        aux = A[i];
                        j = i - 1;
                        while ( (j >= 0) && (A[j] > aux) )
                        {
                                 A[j+1] = A[j];
                                 j--;
                        }
                        A[j+1] = aux;
               }
```