

# Algoritmos de ordenación en C

## 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 principales algoritmos de ordenación interna son:

- **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} <-- Se coloca el 4, el más pequeño, en primera posición : se cambia el 4 por el 40.

{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.

Si el array tiene N elementos, el número de comprobaciones que hay que hacer es de  $N*(N-1)/2$ .

```
void seleccion(int *array,int n)
{
int i,j,menor,aux;

for(i=0;i<n-1;i++)
{
for(j=i+1,menor=i;j<n;j++)
if(array[j]<array[menor]) // Si el elemento j es menor que el menor:
menor=j; // el menor pasa a ser el elemento j.
aux=array[i]; // Se intercambian los elementos
array[i]=array[menor]; // de las posiciones i y menor
array[menor]=aux; // usando una variable auxiliar.
}
}
```

- **Burbuja:** 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.

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 comprobación y hacer una tercera.

Si el array tiene N elementos, para estar seguro de que el array está ordenado, hay que hacer N-1 pasadas, por lo que habría que hacer  $(N-1)*(N-i-1)$  comparaciones.

```
void burbuja(int *array,int n)
{
int i,j,aux;

for(i=0;i<n-1;i++) // Hacer N-1 pasadas.
    for(j=0;j<n-i-1;j++) // Mirar los N-i-1 pares.
        if(array[j+1]<array[j]) // Si el elemento j+1 es menor que el elemento j:
        {
            aux=array[j+1]; // Se intercambian los elementos
            array[j+1]=array[j]; // de las posiciones j y j+1
            array[j]=aux; // usando una variable auxiliar.
        }
}
```

- **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. 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:

{40,40,4,9,10,35} <-- aux=21;  
{21,40,4,9,10,35} <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:

{21,40,40,9,10,35} <-- aux=4;  
{21,21,40,9,10,35} <-- aux=4;  
{4,21,40,9,10,35} <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:

{4,21,40,40,10,35} <-- aux=9;  
{4,21,21,40,10,35} <-- aux=9;  
{4,9,21,40,10,35} <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:

{4,9,21,40,40,35} <-- aux=10;  
{4,9,21,21,40,35} <-- aux=10;

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

Y por último insertamos el 35:

{4,9,10,21,40,40} <-- aux=35;

{4,9,10,21,35,40} <-- El array está ordenado.

En el peor de los casos, el número de comparaciones que hay que realizar es de  $N*(N-1)/2$ .

```
void insercion_directa (int *array,int n)
{
int i,j,aux;

for(i=1;i<n-1;i++) // i contiene el número de elementos de la sublista.
    { // Se intenta añadir el elemento i.
        aux=array[i];
        for(j=i-1;j>=0;j--) // Se recorre la sublista de atrás a adelante para buscar
            { // la nueva posición del elemento i.
                if(aux>array[j]) // Si se encuentra la posición:
                    {
                        array[j+1]=aux; // Ponerlo
                        break; // y colocar el siguiente número.
                    }
                else // si no, sigue buscándola.
                    array[j+1]=array[j];
            }
        if(j==-1) // si se ha mirado todas las posiciones y no se ha encontrado la
correcta
            array[0]=aux; // es que la posición es al principio del todo.
    }
}
```

- **Inserción binaria:** Es el mismo método que la inserción directa, excepto que la búsqueda del orden de un elemento en la sublista ordenada se realiza mediante una búsqueda binaria, lo que supone un ahorro de tiempo (ver algoritmos de búsqueda).
- **Shell:** Es una mejora del método de inserción directa, utilizado cuando el array tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (llamado salto) a su izquierda. Este salto es constante, y su valor inicial es  $N/2$  (siendo  $N$  el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, los pasos para ordenar el array {40,21,4,9,10,35} mediante el método de Shell serían:

Salto=3:

Primera pasada:

{9,21,4,40,10,35} <-- se intercambian el 40 y el 9.

{9,10,4,40,21,35} <-- se intercambian el 21 y el 10.

Salto=1:

Primera pasada:

{9,4,10,40,21,35} <-- se intercambian el 10 y el 4.

{9,4,10,21,40,35} <-- se intercambian el 40 y el 21.

{9,4,10,21,35,40} <-- se intercambian el 35 y el 40.

Segunda pasada:

{4,9,10,21,35,40} <-- se intercambian el 4 y el 9.

Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

```
void shell (int *array,int n)
{
int salto,cambios,aux,i;

for(salto=n/2;salto!=0;salto/=2) // El salto va desde N/2 hasta 1.
  for(cambios=1;cambios!=0;) // Mientras se intercambie algún elemento:
  {
  cambios=0;
  for(i=salto;i<n;i++) // se da una pasada
    if(array[i-salto]>array[i]) // y si están desordenados
    {
      aux=array[i]; // se reordenan
      array[i]=array[i-salto];
      array[i-salto]=aux;
      cambios++; // y se cuenta como cambio.
    }
  }
}
```

- **Ordenación rápida (quicksort):** Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Normalmente se toma como pivote el primer elemento de array, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran. Por ejemplo, para dividir el array {21,40,4,9,10,35}, los pasos serían:

{21,40,4,9,10,35} <-- se toma como pivote el 21. La búsqueda de izquierda a derecha encuentra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote. Se intercambian:

{21,10,4,9,40,35} <-- Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el pivote en su lugar (en el número encontrado por la segunda búsqueda, el 9, quedando:

{9,10,4,21,40,35} <-- Ahora tenemos dividido el array en dos arrays más pequeños: el {9,10,4} y el {40,35}, y se repetiría el mismo proceso.

- **Intercalación:** no es propiamente un método de ordenación, consiste en la unión de dos arrays ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arrays de izquierda a derecha e ir cogiendo el menor de los dos elementos, de forma que sólo aumenta el contador del array del que sale el elemento siguiente para el array-suma. Si quisiéramos sumar los arrays {1,2,4} y {3,5,6}, los pasos serían:

Inicialmente:  $i1=0, i2=0, is=0$ .

Primer elemento: mínimo entre 1 y 3 = 1. Suma={1}.  $i1=1, i2=0, is=1$ .

Segundo elemento: mínimo entre 2 y 3 = 2. Suma={1,2}.  $i1=2, i2=0, is=2$ .

Tercer elemento: mínimo entre 4 y 3 = 3. Suma={1,2,3}.  $i1=2, i2=1, is=3$ .

Cuarto elemento: mínimo entre 4 y 5 = 4. Suma={1,2,3,4}.  $i1=3, i2=1, is=4$ .

Como no quedan elementos del primer array, basta con poner los elementos que quedan del segundo array en la suma:

Suma={1,2,3,4}+{5,6}={1,2,3,4,5,6}

#### Quicksort

La implementación es claramente recursiva, y suponiendo el pivote el primer elemento del array, el programa sería:

```
#include<stdio.h>
```

```
void ordenar(int *,int,int);
```

```
void main()
```

```
{
// Dar valores al array
ordenar(array,0,N-1); // Para llamar a la función
}
```

```
void ordenar(int *array,int desde,int hasta)
```

```
{
int i,d,aux; // i realiza la búsqueda de izquierda a derecha
// y j realiza la búsqueda de derecha a izquierda.
if(desde>=hasta)
return;
```

```
for(i=desde+1,d=hasta;;) // Valores iniciales de la búsqueda.
```

```
{
for(;i<=hasta && array[i]<=array[desde];i++); // Primera búsqueda
for(;d>=0 && array[d]>=array[desde];d--); // segunda búsqueda
if(i<d) // si no se han cruzado:
```

```
{
aux=array[i]; // Intercambiar.
array[i]=array[d];
array[d]=aux;
}
```

```
else // si se han cruzado:
break; // salir del bucle.
```

```
}
if(d==desde-1) // Si la segunda búsqueda se sale del array
```

```
d=desde; // es que el pivote es el elemento
// más pequeño: se cambia con él mismo.
```

```
aux=array[d]; // Colocar el pivote
array[d]=array[desde]; // en su posición.
array[desde]=aux;
```

```
ordenar(array,desde,d-1); // Ordenar el primer array.
ordenar(array,d+1,hasta); // Ordenar el segundo array.
}
```

- En C hay una función que realiza esta ordenación sin tener que implementarla, llamada qsort (incluida en stdlib.h):

```
qsort(nombre_array,numero,tamaño,función);
```

donde nombre\_array es el nombre del array a ordenar, numero es el número de elementos del array, tamaño indica el tamaño en bytes de cada elemento y función es una función, que hay que implementar, que recibe dos elementos y devuelve 0 si son iguales, algo menor que 0 si el primero es menor que el segundo, y algo mayor que 0 si el segundo es menor que el primero. Por ejemplo, el programa de antes sería:

```
#include<stdio.h>
#include<stdlib.h>
```

```
int funcion(const void *,const void *);
```

```
void main()
{
// Dar valores al array
qsort(array,N,sizeof(array[0]),funcion); // Ordena el array.
}
```

```
int funcion(const void *a,const void *b)
{
if(*(int *)a<*(int *)b)
return(-1);
else if(*(int *)a>*(int *)b)
return(1);
else
return(0);
}
```

Claramente, es mucho más cómodo usar qsort que implementar toda la función, pero hay que tener mucho cuidado con el manejo de los punteros en la función, sobre todo si se está trabajando con estructuras.

- **Ejercicio completo con las funciones de ordenación de vectores en C (funcion.h):**

```
#include<stdlib.h>
#include<stdio.h>
#include <time.h>
```

```
void rellenar(int *array, int n)
{
int i;
randomize();
for(i=0;i<n;i++)
array[i]=random(100);
}
```

```
void visualizar(int *array, int n)
{
int i;

for(i=0;i<n;i++)
printf("%4d ",array[i]);
}
```

```

void copiar(int *vector, int n, int *copia)
{
int i;

for(i=0;i<n;i++)
    copia[i]=vector[i];
}

void seleccion(int *array,int n)
{

int i,j,menor,aux;

for(i=0;i<n-1;i++)
{
    for(j=i+1,menor=i;j<n;j++)
        if(array[j]<array[menor]) // Si el elemento j es menor que el menor:
            menor=j; // el menor pasa a ser el elemento j.
    aux=array[i]; // Se intercambian los elementos
    array[i]=array[menor]; // de las posiciones i y menor
    array[menor]=aux; // usando una variable auxiliar.
}
}

void burbuja(int *array,int n)
{
int i,j,aux;

for(i=0;i<n-1;i++) // Hacer N-1 pasadas.
    for(j=0;j<n-i-1;j++) // Mirar los N-i-1 pares.
        if(array[j+1]<array[j]) // Si el elemento j+1 es menor que el elemento j:
            {
                aux=array[j+1]; // Se intercambian los elementos
                array[j+1]=array[j]; // de las posiciones j y j+1
                array[j]=aux; // usando una variable auxiliar.
            }
}

void insercion_directa (int *array,int n)
{
int i,j,aux;

for(i=1;i<n-1;i++) // i contiene el número de elementos de la sublista.
    { // Se intenta añadir el elemento i.
        aux=array[i];
        for(j=i-1;j>=0;j--) // Se recorre la sublista de atrás a adelante para buscar
            { // la nueva posición del elemento i.
                if(aux>array[j]) // Si se encuentra la posición:
                    {
                        array[j+1]=aux; // Ponerlo
                        break; // y colocar el siguiente número.
                    }
                else // si no, sigue buscándola.
                    array[j+1]=array[j];
            }
        if(j==-1) // si se ha mirado todas las posiciones y no se ha encontrado la correcta
            array[0]=aux; // es que la posición es al principio del todo.
    }
}

```

```

    }
}

void shell (int *array,int n)
{

int salto,cambios,aux,i;

for(salto=n/2;salto!=0;salto/=2) // El salto va desde N/2 hasta 1.
    for(cambios=1;cambios!=0;) // Mientras se intercambie algún elemento:
    {
        cambios=0;
        for(i=salto;i<n;i++) // se da una pasada
            if(array[i-salto]>array[i]) // y si están desordenados
            {
                aux=array[i]; // se reordenan
                array[i]=array[i-salto];
                array[i-salto]=aux;
                cambios++; // y se cuenta como cambio.
            }
    }
}

```

- programa que aplica las funciones de ordenación de vectores:

```

#include<stdio.h>
#include "funcion.h"

#define NELEMENTOS 20
void main()
{

int vector[NELEMENTOS],copia[NELEMENTOS];
//*****
rellenar(&vector[0],NELEMENTOS);
printf("\n\n los elementos a ordenar\n");
copiar(&vector[0],NELEMENTOS,&copia[0]);
visualizar(&vector[0],NELEMENTOS);
seleccion(&copia[0],NELEMENTOS);
printf("\n ordenados por algoritmo seleccion.....:\n");
visualizar(&copia[0],NELEMENTOS);
//*****
printf("\n\n los elementos a ordenar\n");
copiar(&vector[0],NELEMENTOS,&copia[0]);
visualizar(&vector[0],NELEMENTOS);
burbuja(&copia[0],NELEMENTOS);
printf("\n ordenados por algoritmo burbuja:\n");
visualizar(&copia[0],NELEMENTOS);
//*****
printf("\n\n los elementos a ordenar\n");
copiar(&vector[0],NELEMENTOS,&copia[0]);
visualizar(&vector[0],NELEMENTOS);
insercion_directa(&copia[0],NELEMENTOS);
printf("\n ordenados por algoritmo insercion_directa:\n");
visualizar(&copia[0],NELEMENTOS);
//*****
printf("\n\n los elementos a ordenar\n");

```

```

copiar(&vector[0],NELEMENTOS,&copia[0]);
visualizar(&vector[0],NELEMENTOS);
shell(&copia[0],NELEMENTOS);
printf("\n ordenados por algoritmo shell:\n");
visualizar(&copia[0],NELEMENTOS);
}

```

**Ejercicio propuesto 1:**

Compare la eficiencia relativa de diferentes métodos de ordenación. Rellene la siguiente tabla donde cada columna recoge un método, y cada fila el número de datos a ordenar. En las casillas en blanco escribiremos el tiempo en milisegundos que le cuesta a cada método ordenar todos esos datos:

	Inserción	Selección	Burbuja	QuickSort
<b>100</b>				
<b>500</b>				
<b>1000</b>				
<b>5000</b>				
<b>10000</b>				
<b>50000</b>				
<b>100000</b>				

No todos los alumnos lograrán rellenar todas las casillas, pues depende del equipo del que dispongan. Escriba una raya en aquellos casos para los que no logre medir el tiempo por la razón que sea.

Tener en cuenta las siguientes sentencias:

```

#include<time.h>
clock_t tini,tfin;
tini=clock();
.....algoritmo de ordenacion
tfin=clock();
printf("\n ordenados por algoritmo X.....tiempoo=%f:\n", (tfin-tini)/1000);

```

**Ejercicio propuesto 2:**

Realizar una un programa que ordene una matriz de dos dimensiones, con cualquiera de los algoritmos anteriores de ordenación de vectores, para ello desarrollad las siguientes funciones;

- void pasar\_a\_vector(int \*, int,int,int \*)

Pasa la dirección y el número de filas y columnas de una matriz y la convierta en un vector cuya dirección conocemos.

- void pasar\_a\_matriz(int\*,int,int,int \*) proceso inverso al anterior.
- void visualizar\_matriz(int \*,int,int)
- void rellenar\_matriz(int \*,int.int)

Texto de algoritmos de ordenación de : [www.algoriemia.net](http://www.algoriemia.net)

Ejercicios resueltos en C: Lola Cano Gil