2012

ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA EN C

Asignatura: Algoritmo y Estructura de Datos

Departamento de Computación – UNAN León

TEMA 1: ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA

1.1 INTRODUCCIÓN:

Uno de los procedimientos más comunes y útiles en el procesamiento de datos, es la clasificación u ordenación de los mismos. Se considera **ordenar** al proceso de reorganizar un conjunto de objetos en una secuencia determinada. El objetivo de este proceso generalmente es facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto.

Ordenar es simplemente **colocar información de una manera especial** basándonos en un criterio de ordenamiento. En la computación el ordenamiento de datos también cumple un rol muy importante, ya sea como un fin en sí o como parte de otros procedimientos más complejos.

1.2 CONCEPTOS PRELIMINARES:

- **Clave:** La parte de un registro por la cual se ordena la lista. Por ejemplo, una lista de registros con campos nombre, dirección y teléfono se puede ordenar alfabéticamente de acuerdo a la clave nombre. En este caso los campos dirección y teléfono no se toman en cuenta en el ordenamiento.
- **Criterio de ordenamiento (o de comparación):** EL criterio que utilizamos para asignar valores a los registros con base en una o más claves. De esta manera decidimos si un registro es mayor o menor que otro.
- **Registro:** Un grupo de datos que forman la lista. Pueden ser datos atómicos (enteros, caracteres, reales, etc.) o grupos de ellos, que en C equivalen a las estructuras.

1.3 ORDENAMIENTO DE DATOS:

Por lo general, todos los algoritmos de ordenación funcionan de una forma similar. Toman una lista de elementos, en nuestro caso un array, comparan sus elementos siguiendo una estrategia definida y, según el resultado de dicha comparación, mueven los datos de un lugar a otro hasta conseguir una lista (array) final ordenado.

La ordenación o clasificación de datos (**sort**, en inglés) es una operación consistente en disponer un conjunto de datos en algún determinado orden con respecto a uno de los campos de elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un campo número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres; los elementos numéricos se pueden ordenar en orden creciente o decreciente de acuerdo al valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina **clave**.

Una colección de datos (estructura) puede ser almacenada en un archivo, un array (vector o tabla), un array de registros, una lista enlazada o un árbol. Cuando los datos están almacenados en un array, una lista enlazada o un árbol, se denomina ***ordenación interna***. Si los datos están almacenados en un archivo, el proceso de ordenación se llama ***ordenación externa.***

Los métodos (algoritmos) de ordenación son numerosos, por ello se debe prestar especial atención en su elección.

1.4 ¿CÓMO SE SABE CUÁL ES EL MEJOR ALGORITMO?:

Cada algoritmo se comporta de modo diferente de acuerdo a la cantidad y la forma en que se le presenten los datos, entre otras cosas. Debes conocer a fondo el problema que quieres resolver, y aplicar el más adecuado. Aunque hay algunas preguntas que te pueden ayudar a elegir:

- **¿Qué grado de orden tendrá la información que vas a manejar?** Si la información va a estar casi ordenada y no quieres complicarte, un algoritmo sencillo como el ordenamiento burbuja será suficiente. Si por el contrario los datos van a estar muy desordenados, un algoritmo poderoso como Quicksort puede ser el más indicado. Y si no puedes hacer una presunción sobre el grado de orden de la información, lo mejor será elegir un algoritmo que se comporte de manera similar en cualquiera de estos dos casos extremos.
- **¿Qué cantidad de datos vas a manipular?** Si la cantidad es pequeña, no es necesario utilizar un algoritmo complejo, y es preferible uno de fácil implementación. Una cantidad muy grande puede hacer prohibitivo utilizar un algoritmo que requiera de mucha memoria adicional.
- **¿Qué tipo de datos quieres ordenar?** Algunos algoritmos sólo funcionan con un tipo específico de datos (enteros, enteros positivos, etc.) y otros son generales, es decir, aplicables a cualquier tipo de dato.
- **¿Qué tamaño tienen los registros de tu lista?** Algunos algoritmos realizan múltiples intercambios (burbuja, inserción). Si los registros son de gran tamaño estos intercambios son más lentos.

Se han desarrollado muchas técnicas en este ámbito, cada una con características específicas, y con ventajas y desventajas sobre las demás. Algunas de las más comunes son:

- **Ordenamiento por el método de la burbuja.**
- **Ordenamiento por el método de inserción.**
- **Ordenamiento por el método de Quicksort.**

1.5 ORDENAMIENO POR MÉTODO DE LA BÚRBUJA (BUBBLESORT):

Hay muchas formas de clasificar datos y una de las más conocidas es la clasificación por el método de la burbuja. Es uno de los más simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.

Veamos a continuación el algoritmo correspondiente, para ordenar una lista de menor a mayor, partiendo de que los datos a ordenar están en una lista de **N** elementos:

1. Comparamos el primer elemento con el segundo, el segundo con el tercero el tercero con el cuarto, etc. Cuando el resultado de una comparación sea "mayor que", se intercambian los valores de los elementos comparados. Con esto conseguimos llevar el valor mayor a la posición **n**.
2. Repetimos el punto 1, ahora para los N-1 primeros elementos de la lista y así sucesivamente.
3. Repetimos el punto 1, ahora para los N-2 primeros elementos de la lista y así sucesivamente.
4. El proceso termina después de repetir el punto 1, N-1 veces, o cuando al finalizar la ejecución del punto 1 no haya habido ningún cambio.

Código en C del algoritmo:

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante entera	Tamaño de la lista
i	Entero	Contador
j	Entero	Contador
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
for (i=0; i<TAM; i++)
 {
   for j=0 ; j<TAM - 1; j++)
   {
            if (lista[j] > lista[j+1])
             {
                  temp = lista[j];
                  lista[j] = lista[j+1];
                  lista[j+1] = temp;
             }
   }
}
```

Veamos con un ejemplo, cómo funciona el algoritmo: Supongamos el siguiente arreglo de enterosa a ordenarse en formato ascendente: **int datos[6] = {40,21,4,9,10,35};**

Primera pasada:
{40, 21, 4, 9, 10, 35}
{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:
{21, 4, 9, 10, 35, 40}
{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.
{4, 9, 10, 21, 35, 40} <-- No hay cambio entre el 21 y el 35.
{4, 9, 10, 21, 35, 40} <-- El 35 y el 40, no se comparan, porque el 40 es el mayor.

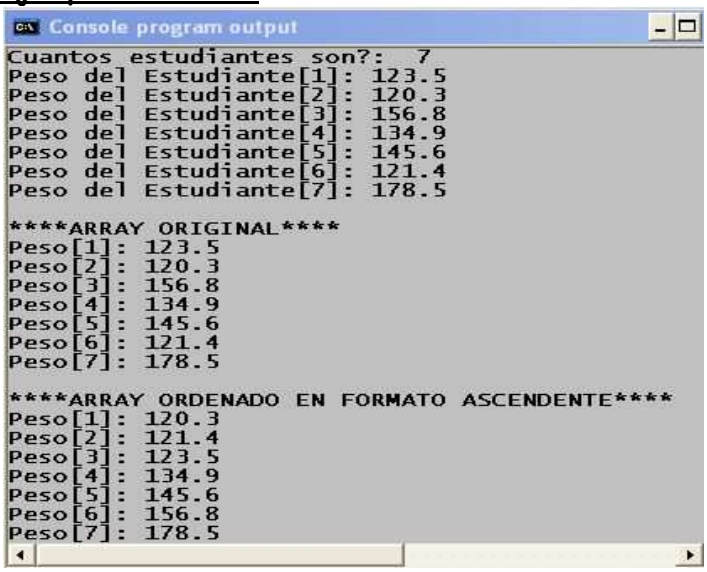
En este ejemplo, los datos ya están ordenados, pero para comprobarlo habría que hacer una tercera, cuarta y quinta comprobación.

Ejemplo #1: Programa en C que permite introducir una lista con los pesos de los N estudiantes de la asignatura de Programación Estructurada e imprime los pesos en formato ascendente (menor a mayor).

```
//burbuja_ascendente.c
#include<stdio.h>
#include<stdlib.h>
void main()
{
        float *pesos,temp;
        int i,j,nest;
        printf("Cuantos estudiantes son?:  ");
        scanf("%d",&nest);
        pesos = (float *) malloc(nest *sizeof(float));
        if(pesos==NULL)
        {
                printf("Insuficiente Espacio de Memoria\n");
                exit(-1); //Salir del Programa
        }
```

```
        for (i = 0; i<nest; i++)
        {
                printf("Peso del Estudiante[%d]: ",i+1);
                scanf("%f",(pesos+i));
        }
        printf("\n****ARRAY ORIGINAL****\n");
        for (i = 0;i<nest;i++)
                        printf("Peso[%d]: %.1f\n",i+1,*(pesos+i));

        /*Ordenación del array con el método de la burbuja en formato
        ascendente*/
        for (i=0;i<nest;i++)
              for(j=0;j<(nest-1);j++)
                    if(pesos[j] > pesos[j+1])
                    {
                          temp=pesos[j];
                          pesos[j] = pesos[j+1];
                          pesos[j+1] = temp;
                    }

        printf("\n****ARRAY ORDENADO EN FORMATO ASCENDENTE****\n");
        for (i = 0;i<nest;i++)
                        printf("Peso[%d]: %.1f\n",i+1,*(pesos+i));
}
```

Ejemplo de Salida:



Ejemplo #2: Programa en C que permite introducir una lista con los pesos de los N estudiantes de la asignatura de Programación Estructurada e imprime los pesos en formato descendente (mayor a menor).

```
//burbuja_descendente.c
#include<stdio.h>
#include<stdlib.h>
void main()
{
        float *pesos,temp;
        int i,j,nest;
        printf("Cuantos estudiantes son?:  ");
        scanf("%d",&nest);
        pesos = (float *) malloc(nest *sizeof(float));
        if(pesos==NULL)
        {
                printf("Insuficiente Espacio de Memoria\n");
                exit(-1); //Salir del Programa
        }

        for (i = 0; i<nest; i++)
        {
                printf("Peso del Estudiante[%d]: ",i+1);
                scanf("%f",(pesos+i));
        }
        printf("\n****ARRAY ORIGINAL****\n");
        for (i = 0;i<nest;i++)
                        printf("Peso[%d]: %.1f\n",i+1,*(pesos+i));
```

```
/*Ordenación del array con el método de la burbuja en formato
descendente*/
for (i=0;i<nest;i++)
        for(j=0;j<(nest-1);j++)
                if(pesos[j] < pesos[j+1])
                {
                        temp=pesos[j];
                        pesos[j] = pesos[j+1];
                        pesos[j+1] = temp;
                }
```

```
        printf("\n****ARRAY ORDENADO EN FORMATO DESCENDENTE****\n");
                for (i = 0;i<nest;i++)
                        printf("Peso[%d]: %.1f\n",i+1,*(pesos+i));
}
```

Ejemplo de Salida:

1.5.1 ANÁLISIS DEL ALGORITMO DE LA BURBUJA:

Éste es el análisis para la versión no optimizada del algoritmo:

- **Estabilidad:** Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es estable.
- **Requerimientos de Memoria:** Este algoritmo sólo requiere de una variable adicional para realizar los intercambios.
- **Tiempo de Ejecución:** El ciclo interno se ejecuta n veces para una lista de n elementos. El ciclo externo también se ejecuta n veces. Es decir, la complejidad es n * n = $O(n^2)$. El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo $O(n^2)$.

Ventajas:

1. Fácil implementación.
2. No requiere memoria adicional.

Desventajas:

1. Muy lento.
2. Realiza numerosas comparaciones.
3. Realiza numerosos intercambios.

Este algoritmo es uno de los más pobres en rendimiento.

1.6 ORDENAMIENTO POR EL MÉTODO DE INSERCIÓN:

Este algoritmo también es bastante sencillo. ¿Has jugado cartas? ¿Cómo las vas ordenando cuando las recibes? Podría ser de esta manera: Se toma la primera y la coloco en mi mano. Luego se toma la segunda y la comparo con la que tengo: si es mayor, la pongo a la derecha, y si es menor a la izquierda (también me fijo en el color, pero omitiremos esa parte para concentrarme en la idea principal). Después tomar la tercera y compararlas con las que se tienen en la mano, desplazándola hasta que quede en su posición final. Continuar haciendo esto, insertando cada carta en la posición que le corresponde, hasta tener todas en orden. ¿Lo haces así tú también? Bueno, pues si es así entonces comprenderás fácilmente este algoritmo, porque es el mismo concepto.

Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos así como así o se perderá un elemento. Lo que se hace es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.

El algoritmo para este método es el siguiente: inicialmente se ordenan los dos primeros elementos del array, luego se inserta el tercer elemento en la posición correcta con respecto a los tres primeros elementos ya clasificados y así sucesivamente hasta llegar al último elemento del array.

Código en C del algoritmo:

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante Entera	Tamaño de la lista
i	Entero	Contador
j	Entero	Contador
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
for (i=1; i<TAM; i++)
{
     temp = lista[i];
     j = i - 1;
     while ( (lista[j] > temp) && (j >= 0) )
     {
          lista[j+1] = lista[j];
          j--;
     }
       lista[j+1] = temp;
}
```

Veamos con un ejemplo, cómo funciona el algoritmo: Supongamos el siguiente arreglo de enterosa a ordenarse en formato ascendente: **int datos[5] = {4,3,5,2,1};**

4 - 3 - 5 - 2 - 1
temp toma el valor del segundo elemento, 3. La primera carta es el 4. Ahora comparamos: 3 es menor que 4. Luego desplazamos el 4 una posición a la derecha y después copiamos el 3 en su lugar.
4 - 4 - 5 - 2 - 1
3 - 4 - 5 - 2 - 1
El siguiente elemento es 5. Comparamos con 4. Es mayor que 4, así que no ocurren intercambios.
Continuamos con el 2. Es menor que cinco: desplazamos el 5 una posición a la derecha:
3 - 4 - 5 - 5 - 1
Comparamos con 4: es menor, así que desplazamos el 4 una posición a la derecha:
3 - 4 - 4 - 5 - 1
Comparamos con 3. Desplazamos el 3 una posición a la derecha:
3 - 3 - 4 - 5 - 1
Finalmente copiamos el 2 en su posición final:
2 - 3 - 4 - 5 - 1
El último elemento a ordenar es el 1. Cinco es menor que 1, así que lo desplazamos una posición a la derecha:
2 - 3 - 4 - 5 - 5
Continuando con el procedimiento la lista va quedando así:
2 - 3 - 4 - 4 - 5
2 - 3 - 3 - 4 - 5
2 - 2 - 3 - 4 - 5
1 - 2 - 3 - 4 - 5

Ejemplo #3: Programa en C que permite introducir una lista con los pesos de los N estudiantes de la asignatura de Programación Estructurada e imprime los pesos ordenados de menor a mayor utilizando el método de inserción.

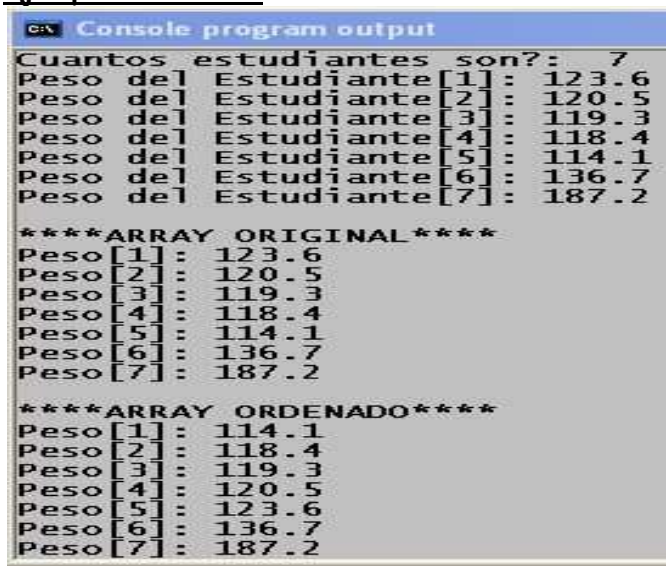
```
//inserción.c
#include<stdio.h>
#include<stdlib.h>
void main()
{
        float *pesos,indice;
        int i,j,nest,e;
        printf("Cuantos estudiantes son?:  ");
        scanf("%d",&nest);
        pesos = (float *) malloc(nest *sizeof(float));
```

```
if(pesos==NULL)
{
        printf("Insuficiente Espacio de Memoria\n");
        exit(-1); //Salir del Programa
}
for (i=0; i<nest; i++)
{
        printf("Peso del Estudiante[%d]: ",i+1);
        scanf("%f",&pesos[i]);
}

printf("\n****ARRAY ORIGINAL****\n");
for (i=0;i<nest;i++)
                printf("Peso[%d]: %.1f\n",i+1,pesos[i]);
```

```
//Ordenación del array a través del método de inserción
for (e=1;e<nest;e++)
 {
       indice = pesos[e];
       j = e-1;
       while (j>=0 && pesos[j]>indice)
       {
              pesos[j+1] = pesos[j];
              j--;
        }
       pesos[j+1]=indice;
}
```

```
printf("\n****ARRAY ORDENADO****\n");
for (i=0;i<nest;i++)
        printf("Peso[%d]: %.1f\n",i+1,pesos[i]);
}
```

Ejemplo de Salida:



Ejemplo #4: Programa en C que permite introducir una lista con los pesos de los N estudiantes de la asignatura de Programación Estructurada e imprime los pesos ordenados de mayor a menor utilizando el método de inserción.

```
//inserción.c
#include<stdio.h>
#include<stdlib.h>
void main()
{
        float *pesos,indice;
        int i,j,nest,e;
        printf("Cuantos estudiantes son?:  ");
        scanf("%d",&nest);
        pesos = (float *) malloc(nest *sizeof(float));
        if(pesos==NULL)
        {
                printf("Insuficiente Espacio de Memoria\n");
                exit(-1); //Salir del Programa
        }

        for (i=0; i<nest; i++)
        {
                printf("Peso del Estudiante[%d]: ",i+1);
                scanf("%f",&pesos[i]);
        }
```

```
printf("\n****ARRAY ORIGINAL****\n");
for (i=0;i<nest;i++)
        printf("Peso[%d]: %.1f\n",i+1,pesos[i]);
```

```
//Ordenación del array a través del método de inserción
for (e=1;e<nest;e++)
 {
      indice = pesos[e];
      j = e-1;
      while (j>=0 && pesos[j]<indice)
      {
            pesos[j+1] = pesos[j];
            j--;
       }
      pesos[j+1]=indice;
}
```

```
printf("\n****ARRAY ORDENADO****\n");
      for (i = 0;i<nest;i++)
            printf("Peso[%d]: %.1f\n",i+1,pesos[i]);
}
```

Ejemplo de Salida:

```
Console program output
Cuantos estudiantes son?:  7
Peso del Estudiante[1]: 123.6
Peso del Estudiante[2]: 120.5
Peso del Estudiante[3]: 119.3
Peso del Estudiante[4]: 118.4
Peso del Estudiante[5]: 114.1
Peso del Estudiante[6]: 136.7
Peso del Estudiante[7]: 187.2

****ARRAY ORIGINAL****
Peso[1]: 123.6
Peso[2]: 120.5
Peso[3]: 119.3
Peso[4]: 118.4
Peso[5]: 114.1
Peso[6]: 136.7
Peso[7]: 187.2

****ARRAY ORDENADO****
Peso[1]: 187.2
Peso[2]: 136.7
Peso[3]: 123.6
Peso[4]: 120.5
Peso[5]: 119.3
Peso[6]: 118.4
Peso[7]: 114.1
```

1.6.1 ANÁLISIS DEL ALGORITMO:

- **Estabilidad:** Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es estable.
- **Requerimientos de Memoria:** Una variable adicional para realizar los intercambios.
- **Tiempo de Ejecución:** Para una lista de n elementos el ciclo externo se ejecuta n-1 veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad $O(n^2)$.

Ventajas:
1 Fácil implementación.
2 Requerimientos mínimos de memoria.

Desventajas:
1 Lento.
2 Realiza numerosas comparaciones.

Este también es un algoritmo lento, pero puede ser de utilidad para listas que están ordenadas o semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

1.7 ORDENAMIENTO POR EL MÉTODO DE QUICKSORT:

El ordenamiento rápido (quicksort en inglés) es un algoritmo basado en la técnica de divide y vencerás. Esta es probablemente la técnica más rápida conocida. Fue desarrollada por Tony Hoare en 1960.

La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar, por lo que se puede considerar que aplica la técnica divide y vencerás. El método es, posiblemente, el más pequeño de código, más rápido, más elegante, más interesante y eficiente de los algoritmos de ordenación conocidos.

El método se basa en dividir los n elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición izquierda, un elemento central denominado pivote o elemento de partición, y una partición derecha. La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

Para dividir la lista en particiones (sublistas) se elige uno de los elementos de la lista y se utiliza como pivote o elemento de partición. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede seleccionar cualquier elemento de la lista como pivote, por

ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial conocido, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo al tamaño relativo de las claves.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra, todos los elementos (claves) mayores que o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo quicksort. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de quicksort es la división o «particionado» recursivo de la lista hasta que todas las sublistas constan de sólo un elemento.

El algoritmo es éste:

- Recorres la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
- Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias.
- Repites esto hasta que se crucen los índices.
- El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

Código en C del algoritmo:

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
inf	Entero	Elemento inferior de la lista
sup	Entero	Elemento superior de la lista
elem_div	El mismo que los elementos de la lista	El elemento divisor
temp	El mismo que los elementos de la lista	Para realizar los intercambios
i	Entero	Contador por la izquierda
j	Entero	Contador por la derechag
cont	Entero	El ciclo continua mientras cont tenga el valor 1

Nombre Procedimiento: OrdRap
Parámetros:
lista a ordenar (lista)
índice inferior (inf)
índice superior (sup)

```
// Inicialización de variables
1. elem_div = lista[sup];
2. i = inf - 1;
3. j = sup;
4. cont = 1;
// Verificamos que no se crucen los límites
5. if (inf >= sup)
        6. retornar;
// Clasificamos la sublista
7. while (cont)
8.      while (lista[++i] < elem_div);
9.      while (lista[--j] > elem_div);
        10. if (i < j)
                11. temp = lista[i];
                12. lista[i] = lista[j];
                13. lista[j] = temp;
        14. else
                15. cont = 0;
                // Copiamos el elemento de división en su posición final
                16. temp = lista[i];
                17. lista[i] = lista[sup];
                18. lista[sup] = temp;
// Aplicamos el procedimiento recursivamente a cada sublista
19. OrdRap (lista, inf, i - 1);
20. OrdRap (lista, i + 1, sup);
```

1.7.1 LOS PASOS QUE SIGUE EL ALGORITMO QUICKSORT:

1. Seleccionar el elemento central de a[0:n-1] como pivote.
2. Dividir los elementos restantes en particiones izquierda y derecha, de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote y que ningún elemento a la derecha tenga una clave más pequeña que la del pivote.
3. Ordenar la partición izquierda utilizando quicksort recursivamente.
4. Ordenar la partición derecha utilizando quicksort recursivamente.
5. La solución es partición izquierda seguida por el pivote y a continuación partición derecha.

Veamos con un ejemplo, cómo funciona el algoritmo: Supongamos el siguiente arreglo de enterosa a ordenarse en formato ascendente: **int datos[7] = {5,3,7,6,2,1,4};**

5 - 3 - 7 - 6 - 2 - 1 - 4
Comenzamos con la lista completa. El elemento divisor será el 4:
5 - 3 - 7 - 6 - 2 - 1 - 4
Comparamos con el 5 por la izquierda y el 1 por la derecha.
5 - 3 - 7 - 6 - 2 - 1 – 4
5 es mayor que cuatro y 1 es menor. Intercambiamos:
1 - 3 - 7 - 6 - 2 - 5 - 4
Avanzamos por la izquierda y la derecha:
1 - 3 - 7 - 6 - 2 - 5 - 4
3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.
1 - 3 - 7 - 6 - 2 - 5 - 4
7 es mayor que 4 y 2 es menor: intercambiamos.
1 - 3 - 2 - 6 - 7 - 5 - 4
Avanzamos por ambos lados:
1 - 3 - 2 - 6 - 7 - 5 - 4
En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[sup] (pasos 16-18):
1 - 3 - 2 - 4 - 7 - 5 - 6
Aplicamos recursivamente a la sublista de la izquierda (índices 0 - 2). Tenemos lo siguiente:
1 - 3 - 2
1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha.
Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[sup]:
1 - 2 - 3

Al llamar recursivamente para cada nueva sublista (lista[0]-lista[0] y lista[2]-lista[2]) se retorna sin hacer cambios (condición 5.).Para resumir te muestro cómo va quedando la lista:

Segunda sublista: lista[4]-lista[6]
7 - 5 - 6
5 - 7 - 6
5 - 6 - 7
Para cada nueva sublista se retorna sin hacer cambios (se cruzan los índices).

Finalmente, al retornar de la primera llamada se tiene el arreglo ordenado:
1 - 2 - 3 - 4 - 5 - 6 - 7

1.7.2 ANÁLISIS DEL ALGORITMO:

- **Estabilidad:** No es estable.
- **Requerimientos de Memoria:** No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila.
- **Tiempo de Ejecución:**

Caso promedio. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:
f(1) = 1
f(n) = n + 2 f(n/2)
La forma cerrada de esta expresión es:
$f(n) = n \log_2 n$
Es decir, la complejidad es $O(n \log_2 n)$.

El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a O(n2).
Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.

Ventajas:
1 Muy rápido
2 No requiere memoria adicional.

Desventajas:
1 Implementación un poco más complicada.
2 Recursividad (utiliza muchos recursos).
3 Mucha diferencia entre el peor y el mejor caso.

La mayoría de los problemas de rendimiento se pueden solucionar con las optimizaciones mencionadas arriba (al costo de complicar mucho más la implementación). Este es un algoritmo que puedes utilizar en la vida real. Es muy eficiente. En general será la mejor opción.

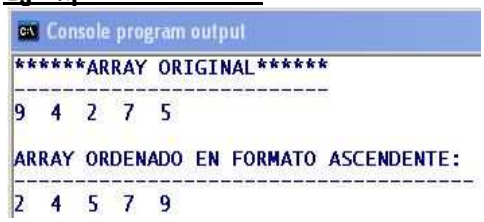
NOTA: En conclusión, se suele recomendar que para listas pequeñas los métodos más eficientes son burbuja y selección; y para listas grandes, el quicksort.

Ejemplo #5: Programa en C que inicializa una lista con valores e imprime los dichos datos ordenados de menor a mayor utilizando el método de quicksort. Nota: Versión Array.

```
/**********Ordencion Quicksort en formato Ascendente. Método recursivo****/
#include<stdio.h>
#include<stdlib.h>
void quicksort(int [], int, int);
void main()
{
        int lista[] = {9,4,2,7,5},i,n_elementos;
        n_elementos = sizeof(lista)/sizeof(int);
        printf ("******ARRAY ORIGINAL******\n-------------------------\n");
        for (i = 0; i < n_elementos; i++)
                printf("%d  ",lista[i]);
        quicksort(lista,0,n_elementos-1);
        printf ("\n\nARRAY ORDENADO EN FORMATO ASCENDENTE: \n");
        printf ("-------------------------------------\n");
        for (i = 0; i < n_elementos; i++)
                printf("%d  ",lista[i]);
}

//Función Quicksort en formato ascendente
void quicksort(int lista[], int inf, int sup)
{
        int mitad, x,izq, der;
        izq = inf; der = sup;
        mitad = lista[(izq + der)/2];
        do
        {
                while(lista[izq] < mitad && izq < sup)
                        izq++;
                while(mitad < lista[der] && der > inf)
                        der--;
                if(izq <= der)
                {
                        x = lista[izq],
                        lista[izq] = lista[der],
                        lista[der] = x;
                        izq++;
                        der--;
                }
```

```
        }while(izq <= der);

        if(inf<der)
                quicksort(lista, inf,der); //mismo proceso con sublista izqda
        if(izq<sup)
                quicksort(lista,izq,sup); //mismo proceso con sublista derecha
}
```

Ejemplo de Salida:

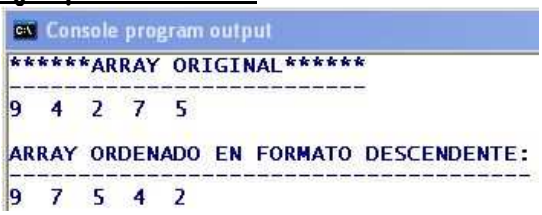


Ejemplo #6: Programa en C que inicializa una lista con valores e imprime los dichos datos ordenados de mayor a menor utilizando el método de quicksort. Nota: Versión Array.

```
/************* Ordencion Quicksort Descendente. Método recursivo***************/
#include<stdio.h>
#include<stdlib.h>
void quicksort(int [], int, int);
void main()
{
        int lista[] = {9,4,2,7,5},i,n_elementos;
        n_elementos = sizeof(lista)/sizeof(int);

        printf ("******ARRAY ORIGINAL******\n-------------------------\n");
        for (i = 0; i < n_elementos; i++)
                printf("%d  ",lista[i]);

        quicksort(lista,0,n_elementos-1);

        printf ("\n\nARRAY ORDENADO EN FORMATO ASCENDENTE: \n");
        printf ("-------------------------------------\n");
        for (i = 0; i < n_elementos; i++)
                printf("%d  ",lista[i]);
}
```

```
//////////////Quicksort Descendente//////////////////////
void quicksort(int lista[], int inf, int sup)
{
        int mitad, x,izq, der;
        izq = inf; der = sup;
        mitad = lista[(izq + der)/2];
        do
        {
                while(lista[izq] > mitad && izq < sup)
                        izq++;
                while(mitad > lista[der] && der > inf)
                        der--;
                if(izq <= der)
                {
                        x = lista[izq],
                        lista[izq] = lista[der],
                        lista[der] = x;
                        izq++;
                        der--;
                }
        }while(izq <= der);

        if(inf<der)
                quicksort(lista, inf,der); //mismo proceso con sublista izqda
        if(izq<sup)
                quicksort(lista,izq,sup); //mismo proceso con sublista derecha
}
```

Ejemplo de Salida:

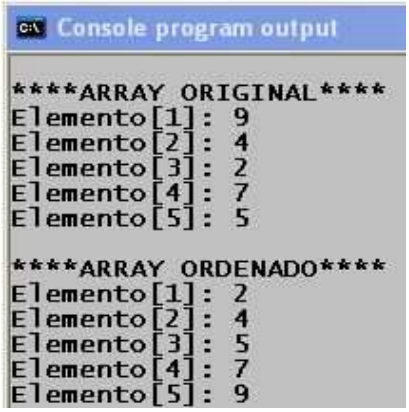


Ejemplo #7: Programa en C que inicializa una lista con valores e imprime los dichos datos ordenados de menor a mayor utilizando el método de quicksort. Nota: Versión Punteros.

```
//metodo_quicksort.c
#include<stdio.h>
#include<stdlib.h>
void quicksort(int *izq, int *der);
void Intercambio(int *a, int *b);
void main()
{
	int lista[]={9,4,2,7,5};
	int i,nelem;
	nelem= sizeof(lista)/sizeof(int);

	printf("\n****ARRAY ORIGINAL****\n");
	for (i=0;i<nelem;i++)
			printf("Elemento[%d]: %d\n",i+1,lista[i]);

	/*Se llama con: quicksort(&vector[0],&vector[n-1]);*/
	quicksort(&lista[0],&lista[nelem-1]);

	printf("\n****ARRAY ORDENADO****\n");
	for (i=0;i<nelem;i++)
		printf("Elemento[%d]: %d\n",i+1,lista[i]);
}

//Ordenación del array a través del método de quicksort
void quicksort(int *izq, int *der)
{
	if(der < izq)
		return;
	int pivot = *izq;
	int *ult = der;
	int *pri = izq;

	while(izq<der)
	{
		while(*izq <= pivot && izq < (der+1))
			izq++;
		while(*der > pivot && der > (izq-1))
			der--;
```

```
            if( izq < der)
                Intercambio(izq,der);
        }
        Intercambio(pri,der);
        quicksort(pri,der-1);
        quicksort(der+1,ult);
}

void Intercambio(int* a, int* b)
{
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

Ejemplo de Salida:



1.8 BÚSQUEDA DE DATOS:

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros, y por ello será necesario determinar si un array contiene un valor que coincida con un cierto valor clave. El proceso de encontrar un elemento específico de un array se denomina búsqueda. En esta sección se examinarán dos técnicas de búsqueda: **búsqueda lineal o secuencial**, la técnica más sencilla, y **búsqueda binaria o dicotómica**, la técnica más eficiente.

1.8.1 BÚSQUEDA SECUENCIAL:

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado clave. En una búsqueda secuencial (a veces llamada búsqueda lineal), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro.

La búsqueda secuencial es necesaria, por ejemplo, si se desea encontrar la persona cuyo número de teléfono es 2315-5678 en un directorio o listado telefónico de su ciudad. Los directorios de teléfonos están organizados alfabéticamente por el nombre del abonado en lugar de por números de teléfono, de modo que deben explorarse todos los números, uno después de otro, esperando encontrar el número.

El algoritmo de búsqueda secuencial compara cada elemento del array con la clave de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados. La eficiencia de la búsqueda secuencial es pobre, tiene complejidad lineal, O(n).

El pseudocódigo puede ser:

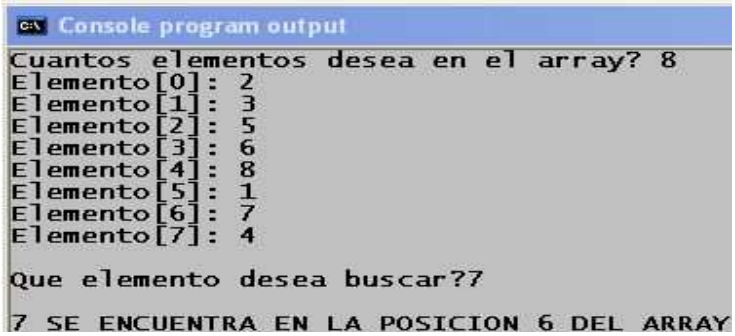
```
<función búsqueda_S(array a, valor que queremos buscar)>
        i=0
        DO WHILE(no encontrado)
                IF(valor = a[i])
                        encontrado
                ENDIF
                i = i + 1
        ENDDO
END<Busqueda_S>
```

Ejemplo #8: Programa en C que implementa la búsqueda secuencial de un elemento determinado dentro de un array de N elementos.

```
//busqueda_secuencial.c
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

int BusquedaSecuencial(int *parray,int buscar,int nelem);

void main()
{
        int *pdatos, nelem, dbuscar,d,result;

        printf("Cuantos elementos desea en el array? ");
        scanf("%d",&nelem);
```

```
        pdatos= (int *) malloc(nelem * sizeof(int));

        if(pdatos==NULL)
        {
                printf("Insuficiente Espacio de Memoria");
                exit(-1);
        }
        for(d=0; d < nelem;d++)
        {
                printf("Elemento[%d]: ",d);
                scanf("%d",(pdatos+d));
        }

        printf("\nQue elemento desea buscar?");
        scanf("%d",&dbuscar);
        //Llamada a la Función que realiza la búsqueda
        result = BusquedaSecuencial(pdatos,dbuscar,nelem);
        if (result != -1)
                printf ("\n%d SE ENCUENTRA EN LA POSICION %d DEL
                ARRAY\n",dbuscar, result);
        else
                printf ("\n %d NO SE ENCUENTRO\n",dbuscar);
}

int BusquedaSecuencial(int *parray,int buscar,int elem)
{
        int i;
        for (i=0; i<elem; i++)
        {
                if (*(parray+i) == buscar)
                        return (i); // Retornamos el índice del valor encontrado
        }
        return (-1);
}
```

Ejemplo de Salida:



1.8.2 BÚSQUEDA BINARIA:

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la búsqueda binaria proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con «J» y se está en la «L» se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Se selecciona el elemento del centro o aproximadamente del centro del array. Si el valor a buscar no coincide con el elemento seleccionado y es mayor que él, se continúa la búsqueda en la segunda mitad del array. Si, por el contrario, el valor a buscar es menor que el valor del elemento seleccionado, la búsqueda continúa en la primera parte del array. En ambos casos, se halla de nuevo el elemento central, correspondiente al nuevo intervalo de búsqueda, repitiéndose el ciclo. El proceso se repite hasta que se encuentra el valor a buscar, o bien hasta que el intervalo de búsqueda sea nulo, lo que querrá decir que el elemento buscado no figura en el array.

Ejemplo #9: Programa en C que implementa la búsqueda binaria de un elemento determinado dentro de un array de N elementos.

```
//busqueda_binaria.c
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

int BusquedaBinaria(int datos[], int nelem, int clave);
```

```
void main()
{
        int *pdatos, nelem, dbuscar,d,result;

        printf("Cuantos elementos desea en el array? ");
        scanf("%d",&nelem);
        pdatos= (int *) malloc(nelem * sizeof(int));
        if(pdatos==NULL)
        {
                printf("Insuficiente Espacio de Memoria");
                exit(-1);
        }
        for(d=0; d < nelem;d++)
        {
                printf("Elemento[%d]: ",d);
                scanf("%d",(pdatos+d));
        }
        printf("\nQue elemento desea buscar?");
        scanf("%d",&dbuscar);
        //Llamada a la Función que realiza la búsqueda
        result = BusquedaBinaria(pdatos,nelem,dbuscar);
        if (result != -1)
                printf ("\n%d SE ENCUENTRA EN LA POSICION %d DEL
                ARRAY\n",dbuscar,result);
        else
                printf ("\n %d NO SE ENCUENTRO\n",dbuscar);
}

int BusquedaBinaria(int lista[], int n, int clave)
{
        int central, bajo, alto;
        int valorCentral;
        bajo = 0;
        alto = n-1;

        while (bajo <= alto)
        {
                central = (bajo+alto)/2; /* índice de elemento central */
                valorCentral = lista[central]; /* valor del índice central */

                if (clave == valorCentral)
                        return (central); /* encontrado, devuelve posición */
```

```
            else if (clave < valorCentral)
                    alto = central-1; /* ir a sublista inferior */
            else
                    bajo = central+1; /* ir a sublista superior */
        }
        return (-1); /* elemento no encontrado */
}
```

Ejemplo #9.1: Programa en C que implementa la búsqueda binaria de un elemento determinado dentro de un array (ordenado en formato ascendente) de N elementos.

```
//busqueda_binaria.c
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
int BusquedaBinaria(int datos[], int nelem, int clave);
void OrdenarBurbuja (int datos[],int nelem)
{
        int i,j,temp;
for (i=0;i<nelem;i++)
        for(j=0;j<(nelem-1);j++)
                if(datos[j] > datos[j+1])
                {
                        temp=datos[j];
                        datos[j] = datos[j+1];
                        datos[j+1] = temp;
                }
}

void main()
{
        int *pdatos, nelem, dbuscar,d,result;

        printf("Cuantos elementos desea en el array? ");
        scanf("%d",&nelem);
        pdatos= (int *) malloc(nelem * sizeof(int));
        if(pdatos==NULL)
        {
                printf("Insuficiente Espacio de Memoria");
                exit(-1);
        }
```

```
	for(d=0; d < nelem;d++)
	{
		printf("Elemento[%d]: ",d);
		scanf("%d",(pdatos+d));
	}
	OrdenarBurbuja(pdatos,nelem);
	printf("\nQue elemento desea buscar?");
	scanf("%d",&dbuscar);
	//Llamada a la Función que realiza la búsqueda
	result = BusquedaBinaria(pdatos,nelem,dbuscar);
	if (result != -1)
		printf ("\n%d SE ENCUENTRA EN LA POSICION %d DEL
ARRAY\n",dbuscar,result);
	else
		printf ("\n %d NO SE ENCUENTRO\n",dbuscar);
}

int BusquedaBinaria(int lista[], int n, int clave)
{
	int central, bajo, alto;
	int valorCentral;
	bajo = 0;
	alto = n-1;
	while (bajo <= alto)
	{
		central = (bajo+alto)/2;  //índice de elemento central
		valorCentral = lista[central]; // valor del índice central
		if (clave == valorCentral)
			return (central); //encontrado, devuelve posición
		else if (clave < valorCentral)
			alto = central-1; //ir a sublista inferior
		else
			bajo = central+1; //ir a sublista superior
	}
	return (-1);  //elemento no encontrado
}
```

EJERCICIOS PROPUESTOS:

1. **Partiendo del siguiente array encuentre el número de pasadas que realiza el algoritmo de ordenación de burbuja para su ordenación.**

47	3	21	32	56	92

2. **Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?**

3	13	8	25	45	23	98	58

3. **Partiendo del siguiente array encuentre las particiones e intercambios que realiza el algoritmo de ordenación quicksort para su ordenación.**

8	43	17	6	40	16	18	97	11	7

4. **Realice un programa que permita ordenar de menor a mayor o de mayor a menor un array con los sueldos de los N empleados de una empresa por cualquiera de los algoritmos de ordenación estudiados.**

```
Console program output
1.- Leer los datos del array
2.- Ordenar de menor a mayor por el Metodo de la Burbuja
3.- Ordenar de mayor a menor por el Metodo de la Burbuja
4.- Ordenar de menor a mayor por el Metodo de Insercion
5.- Ordenar de mayor a menor por el Metodo de Insercion
6.- Ordenar de menor a mayor por el Metodo de QuickSort
7.- Ordenar de mayor a menor por el Metodo de QuickSort
8.- Salir

Introduce una opcion:
```