ORDENACION, BUSQUEDA Y OTROS ALGORITMOS BASICOS

Las operaciones de búsqueda y ordenación son básicas en la programación.

De hecho, muchas actividades humanas requieren que una determinada colección de elementos esté organizada en un orden específico.

A la hora de diseñar un método de búsqueda u ordenación siempre se busca el procedimiento que sea más eficiente.

- Las operaciones de búsqueda y ordenación se clasifican en:
  - **Operaciones internas**, se pueden realizar íntegramente en memoria principal.
  - **Operaciones externas**, donde los datos están almacenados en un dispositivo de almacenamiento masivo. Por el gran tamaño del conjunto, se necesita realizar transferencias de información entre la memoria principal y el dispositivo.

Complejidad de un algoritmo

- Medida del trabajo que realiza así como de los recursos que ha necesitado.
- El estudio de la complejidad, se hace en cuanto a los recursos consumidos (*complejidad espacial*) y el tiempo necesario para su ejecución (*complejidad temporal*). Sin embargo, esta medida valora también el compilador y la máquina donde se ejecute.
- El **orden de magnitud** nos da una medida aproximada de la complejidad, que es más conveniente que obtener medidas precisas.

Por ejemplo si un algoritmo ejecuta dos operaciones P y Q equivalentes en cuanto a complejidad, y si P se realiza N^2 veces y Q se ejecuta N veces, la complejidad sería N^2*P+N*Q. Al ser P y Q equivalentes, el orden de complejidad es N^2.

Búsqueda (interna)

- Dado un conjunto de elementos de un cierto tipo, la búsqueda consiste en determinar si un elemento se encuentra en el conjunto o no.

- Existen diferentes algoritmos de búsqueda y la elección depende de la forma en que se encuentren organizados los datos.

- Se deben tener en cuenta ciertos hechos.
    - Conocer la disposición de los elementos. Puede ocurrir que:
        - Se ignora la disposición de los datos o se sabe que están al azar.
        - Se sabe que los datos están ordenados.

- Conocer si la estructura que almacena los datos es de acceso secuencial o aleatorio.

Búsqueda secuencial

- La más simple ya que no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio.
- Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.
- Se supondrán los datos almacenados en un array y se asumirá acceso secuencial.
- Consideraremos dos métodos: con y sin centinela.

Búsqueda sin centinela

```
#define TAM 100

void SecSin(int CB[TAM], int dato)
{
int i=0;

while ((CB[i]!=dato) && (i<TAM))
  i++;

if (CB[i]==dato) printf("Posicion %d\n",i);
else printf("Elemento no esta en el array");
}
```

- Análisis del algoritmo:
  - Número de iteraciones: Mejor caso=1, Peor caso=TAM, Promedio=(TAM+1)/2. El orden de complejidad es lineal (*O(TAM)*).
  - Cada iteración necesita una suma, dos comparaciones y un `AND lógico`.

- Ejemplo (buscar 8):

```
0 9 5 5 8 4 6 0 4 9
- - - - -
Posicion 5
```

Búsqueda con centinela

- Si tuviésemos la seguridad de que el elemento buscado está en el conjunto, nos evitaría controlar si se supera el límite superior.
- Se almacena un elemento adicional (centinela), que coincidirá con el elemento buscado. De esta forma se asegura que encontraremos el elemento buscado.

```
#define TAM 101

void SecSin(int CB[TAM], int dato)
{
int i=0;

CB[TAM-1]=dato;

while (CB[i]!=dato)
  i++;

if (i==TAM-1) printf("Elemento no esta en el array");
else printf("Posicion %d\n",i);

}
```

- Si el índice alcanza el valor TAM-1, el elemento no se encuentra en el array.
- Una suma y una única comparación (se ahorra una comparación y un and). El algoritmo es más eficiente.

Búsqueda binaria o dicotómica

- Es un método muy eficiente que tiene varios prerequisitos:
  - El conjunto de búsqueda está ordenado.
  - Se dispone de acceso aleatorio.
- Se compara el dato buscado con el elemento central. Según sea menor o mayor se prosigue la búsqueda con el subconjunto anterior o posterior respectivamente al elemento central y así sucesivamente.

```
#define TAM 100

void BusBin(int CB[TAM], int dato)
{
int ini=0,sup=TAM-1,mitad;

mitad=(ini+fin)/2;

while ((ini<=fin)&&(CB[mitad]!=dato))
  {
  if (dato < CB[mitad])
    fin=mitad-1;
  else ini=mitad+1;
  mitad=(ini+fin)/2;
  }

if (dato==CB[mitad]) printf("Posicion %d\n",i);
else printf("Elemento no esta en el array");
```

```
}
```

Análisis del algoritmo:

- Caso más favorable (dato es el elemento mitad): 1 iteración.
- Caso más desfavorable, el número de iteraciones es el menor entero K que verifica $2^K >= TAM$. Esto es, el orden de complejidad es $O(log_2 TAM)$.

- Ejemplo 1 (buscar 8):

```
1 2 2 2 4 4 5 6 6 9  (ini,fin)       mitad
        -            (0  ,9)          4
1 2 2 2 4 4 5 6 6 9
              -      (5  ,9)          7
1 2 2 2 4 4 5 6 6 9
                -    (8  ,9)          8
1 2 2 2 4 4 5 6 6 9
                  -  (9 ,9)           9
No se encontro....   (9 ,8)
```

- Ejemplo 2 (buscar 8):

```
0 1 1 2 2 6 7 7 8 9  (ini,fin)       mitad
          -          (0  ,9)          4
0 1 1 2 2 6 7 7 8 9
                 -    (5  ,9)          7
0 1 1 2 2 6 7 7 8 9
                   -  (8  ,9)          8
Posicion 8
```

Comparación de la búsqueda binaria con la secuencial

Número de elementos examinados (peor caso)		
Tamaño del array	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1000	11	1000
5000	14	5000
100000	18	100000
1000000	21	1000000

Ordenación

- Consiste en poner un conjunto de elementos en un orden ascendente o descendente.

- Es útil...
  - Para la búsqueda.
  - En problemas que requieran tratar todos los elementos con la misma identificación.
  - Si se dispone de estructuras con datos equivalentes, el orden facilitará la mezcla (emparejar elementos de la primera con los de la segunda).

- El objetivo de los algoritmos de ordenación será permutar los elementos del conjunto hasta conseguir un orden creciente o decreciente.

- Existen muchos algoritmos de ordenación con diferentes ventajas e inconvenientes.

- Los procesos básicos de la ordenación son la comparación y el intercambio.

- Un método de ordenación es estable si los elementos iguales (según el criterio de ordenación) aparecen en el mismo orden que en el conjunto inicial.

Método de la búrbuja

- Se basa en recorrer el array (pasada) un cierto número de veces, comparando pares de valores que ocupan posiciones consecutivas (0-1,1-2, ...).

- Si no están ordenados, se intercambian.

- Al final de la primera pasada el elemento mayor estará en la última posición, en la segunda, el segundo elemento llegará a la penúltima y así sucesivamente.

```
#define TAM 100

void Burb1(int CB[TAM])
{
int e,i,aux;

for (e=0;e<(TAM-1);e++)
 for (i=0;i<(TAM-e-1);i++)
    if (CB[i]>CB[i+1])
        {
              aux=CB[i];
              CB[i]=CB[i+1];
              CB[i+1]=aux;
        }
}
```

Ejemplo:

```
4 0 6 5 7 7 0 2 9 7
0 4 5 6 7 0 2 7 7 9
0 4 5 6 0 2 7 7 7 9
0 4 5 0 2 6 7 7 7 9
0 4 0 2 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
```

- Si con cada pasada se lleva un elemento a su posición final, serán necesarias TAM-1 pasadas. El orden de magnitud es aproximadamente TAM^2.

Burbuja mejorada, introduce una mejora considerando la posibilidad de que el conjunto esté ordenado en algún punto del proceso. Si el bucle interno no necesita realizar ningún intercambio en alguna pasada, el conjunto estará ya ordenado.

```
#define TAM 100
void Burb1(int CB[TAM])
{
int e,i,aux,intercambio;
for (e=0;e<(TAM-1);e++)
{
intercambio=0;
 for (i=0;i<(TAM-e-1);i++)
    if (CB[i]>CB[i+1])
        {
              aux=CB[i];
              CB[i]=CB[i+1];
              CB[i+1]=aux;
         intercambio=1;
        }
if (intercambio==0) break;
}
}
```

Ejemplo

```
4 0 6 5 7 7 0 2 9 7
0 4 5 6 7 0 2 7 7 9
0 4 5 6 0 2 7 7 7 9
0 4 5 0 2 6 7 7 7 9
0 4 0 2 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
```

- En el mejor caso (si ya está ordenado) realiza TAM-1 comparaciones.
- En el peor caso (máximo desorden) se necesitan las mismas pasadas que antes y el orden es TAM^2.
- En el caso medio el orden es proporcional a $TAM^2/2$.

Método de selección

- El array se considera formado por 2 partes:
  - una parte ordenada (la izquierda) que estará vacía al principio y al final contendrá todo el array.
  - una parte desordenada (la derecha) que contendrá al principio todo el array y al final estará vacía.

- El proceso consiste en tomar elementos de la parte derecha y colocarlos en la parte izquierda.

- Se selecciona el menor elemento de la parte desordenada y se intercambia con el que ocupa su posición en la otra parte.

- Para ello...
  - En la primera iteración se busca el menor elemento y se intercambia con el que ocupa la posición 0.
  - En la segunda, se busca el menor elemento entre la posición 1 y el final y se intercambia con el elemento en la posición 1. De esta manera las dos primeras posiciones del array están ordenadas y contienen los dos elementos menores dentro del array.
  - Continuar con este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la lista e intercambiándolos adecuadamente.

- En cada pasada se coloca un elemento en su lugar, y la variable e marca donde empezar la búsqueda en la parte desordenada, que será secuencial si no tenemos más información.

- Comienza suponiendo que el elemento e es el menor. Se comprueba la hipótesis comparándolo con cada uno de los restantes. Si se encuentra uno menor, se intercambia.

```
#define TAM 100

void Selec(int CB[TAM])
{
int e,i,PosMenor,aux;

for (e=0; e<(TAM-1) ; e++)
  {
   PosMenor=e;
   for (i=e+1;i<TAM;i++)
    if (CB[i]<CB[PosMenor])
        PosMenor=i;

   aux=CB[e];
```

```
    CB[e]=CB[PosMenor];
     CB[PosMenor]=aux;
    }
  }
```

- Ejemplo:

```
0     1 6 8 5 9 3 0 3 7
0 0     6 8 5 9 3 1 3 7
0 0 1     8 5 9 3 6 3 7
0 0 1 3     5 9 8 6 3 7
0 0 1 3 3     9 8 6 5 7
0 0 1 3 3 5     8 6 9 7
0 0 1 3 3 5 6     8 9 7
0 0 1 3 3 5 6 7     9 8
0 0 1 3 3 5 6 7 8     9
```

- Análisis del algoritmo:
  - El número de comparaciones es independiente de la ordenación inicial.

- El bucle interno hace TAM-1 comparaciones la primera vez, TAM-2 la segunda,..., y 1 la última. El bucle externo hace TAM-1 búsquedas.
- El total de comparaciones es (TAM^2-TAM)/2. Por tanto el orden de complejidad es cuadrático (*$O(TAM^2)$*).

Método de inserción

- Se utiliza un método similar al anterior, tomando un elemento de la parte no ordenada (el primero y no el mínimo como antes) para colocarlo en su lugar en la parte ordenada.

- El primer elemento del array (CB[0]) se considerado ordenado (la lista inicial consta de un elemento).

- Se inserta el segundo elemento (CB[1]) en la posición correcta (delante o detrás de CB[0]) dependiendo de que sea menor o mayor que CB[0].

- Así sucesivamente se va colocando cada elemento en la posición correcta. El proceso se repetirá TAM-1 veces.

- Para colocar el dato en su lugar, se debe
  - Encontrar la posición que le corresponde en la parte ordenada.
  - Hacerle un hueco de forma que se pueda insertar.

- Para encontrar la posición se puede hacer una búsqueda secuencial desde el principio del conjunto hasta encontrar un elemento mayor que el dado.

- Para hacer el hueco hay que desplazar los elementos involucrados una posición a la derecha.

```
#define TAM 100
 void Inserc(int CB[TAM])
 {
int e,i,k,temp;
for (e=1;e<TAM;e++)
{
temp=CB[e];/* temp es el elemento a insertar en el
array ordenado */
i=0;
while (CB[i]<=temp)  /*  buscar  posicion  que  le
corresponde en el array ordenado */
      i++;
if (i<e){
     for (k=e;k>i;k--)
     CB[k]=CB[k-1];
     CB[i]=temp;
     }
```

```
}
}
```

- Ejemplo:

```
4   1 6 9 1 0 2 9 8 4
1 4   6 9 1 0 2 9 8 4
1 4 6   9 1 0 2 9 8 4
1 4 6 9   1 0 2 9 8 4
1 1 4 6 9   0 2 9 8 4
0 1 1 4 6 9   2 9 8 4
0 1 1 2 4 6 9   9 8 4
0 1 1 2 4 6 9 9   8 4
0 1 1 2 4 6 8 9 9   4
0 1 1 2 4 4 6 8 9 9
```

- Análisis del algoritmo:
  - Se puede demostrar que el orden de complejidad es cuadrático ($O(TAM^2)$).

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.

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, intercambiar */
     {
       aux=array[i];
       array[i]=array[d];
       array[d]=aux;
     }
     else              /* si se han cruzado, salir del bucle */
       break;
   }
   if(d==desde-1)
/* Si la segunda búsqueda se sale del array es que el
    pivote es el elemento más pequeño: se cambia con él mismo
*/
     d=desde;

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

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

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

qsort(nombre_array,número,tamaño,función);

donde nombre_array es el nombre del array a ordenar, número es el número de elementos del array, tamaño indica el tamaño en bytes de cada elemento y función es un puntero a 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);
}
```

```
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.