Patrones de resolución con vectores

En este documento comenzamos a trabajar con patrones de resolución de algoritmos, utilizando las estructuras de datos arreglos y registros.

Temas a desarrollar

- Patrones de recorrido
- Patrones de Búsqueda
- Patrones de Ordenamiento

Antes de comenzar con la ejercitación hagamos un resumen

Las estructuras o registros permiten agrupar datos de diversos tipos asociados a una entidad en particular. Por ejemplo las personas tienen documento, nombre, apellido, fecha de nacimiento, etc. Es muy útil para una buena comprensión del código y permite el pasaje de parámetros en forma clara y ordenada.

Los arreglos son estructuras de datos homogéneas (todos sus datos son del mismo tipo) que permiten almacenar un determinado número de datos bajo un mismo identificador, para luego referirse a los mismo utilizando uno o más subíndices. Los arreglos pueden pensarse como vectores, matrices, etc. Para poder utilizar un arreglo, primero es obligatorio su dimensionamiento; es decir, definirlo declarando los rangos de sus subíndices, lo cual determina cuántos elementos se almacenarán y cómo se accederá a los mismos. Todas las celdas de un arreglo son del mismo tipo, este puede ser un tipo simple (int, float, string, char, etc), un tipo complejo definido por el desarrollador (otro arreglo, una estructura, etc)

Hay que tener en cuenta que se utilizan arreglos, de 1 o 2 dimensiones, toda vez que se debe mantener en memoria varios datos de una misma característica, y se accede a los mismos a través de un índice.
El índice varía entre 0 y la dimensión del arreglo menos 1.

Los arreglos son estructuras estáticas, por lo tanto se debe definir durante el desarrollo del programa.

Es importante comenzar a pensar qué tipos de funciones generales se necesitan para utilizar arreglos.

Algunas de estas –las más clásicas- ya las estuvimos trabajando: inicializar; recorrer

Otras las comenzaremos a trabajar ahora: buscar; ordenar

La idea general es pensar y escribir aquellas funciones que se utilizan habitualmente, de forma tal que estén utilizables a la hora de resolver problemas.

Para clarificar veamos con un ejemplo.

Supongamos que debemos procesar las ventas de un comercio, para lo cual se dispone de todos los artículos (código de identificación, descripción, tamaño, precio) en un vector de registros. El problema consiste en facturar a un cliente, para lo cual se ingresa desde el teclado los códigos de identificación y por cada uno de estos se debe **buscar** en el vector a qué artículo corresponde, obtener la descripción y el precio e ir sumado el total.

Si comenzamos a elaborar la solución algorítmica, nos vamos a dar cuenta que dentro del problema planteado hay un "subproblema" que es **¿cómo ese hace la búsqueda?**

Nos podríamos preguntar también **¿es conveniente que el vector esté ordenado?**

Entonces, si pensamos y escribimos previamente estas funciones, nos podremos dedicar a pensar el problema de la facturación .

Funciones de búsqueda

Función de búsqueda

Pensemos ahora cómo buscar en un vector. En primer lugar pensemos en hacer una función (que sea reutilizable), con los parámetros necesarios y suficientes para que pueda ser utilizada en diversos contextos.

"Lamentablemente" el paradigma de programación procedural (el que estamos utilizando) no es muy versátil en cuanto a que una función se pueda utilizar en cualquier contexto sin tener que "copiar y pegar". ¿Qué quiere decir esto? Si escribimos una función para buscar en un vector de números enteros y valor determinado, cuando la queramos utilizar para números reales, deberemos hacer una copia para este nuevo tipo de datos.

Igualmente C++ trabaja con un concepto de "TEMPLATES" que veremos más adelante y permite así que la misma función se utilice con diversos tipos de datos y hasta con algunos cambios funcionales (lo hablamos cuando estemos ordenando vectores)

Comencemos a pensar cómo buscar. Después de pensar un rato, puede que no se me ocurra, o que lo vea muy fácil. Y si, **si no me enrosco es una tarea relativamente fácil**. Es como buscar una prenda en los cajones de la ropa: me fijo en el primero, no está, me fijo en segundo, no está y así sucesivamente hasta encontrarla o llegar al último cajón. Ah!. Entonces puede que no esté. Y si quiero buscar un nombre en una lista de nombres (por ejemplo en una hoja con un nombre en cada renglón). Podemos mirar la primer línea, luego la segunda y así sucesivamente.

Para pensar….¿ayuda en algo que la lista esté ordenada?
¿siempre encontramos lo que buscamos?
¿se tarda mucho en buscar si la lista es muy grande?

¿Ayuda a que esté ordenada?, pensemos un poco acerca de cómo buscamos…. BINGO…. Tendremos que mirar uno a uno los datos hasta……¿encontrar lo que buscamos?. Justamente esta es la pregunta siguiente, NO SIEMPRE ENCONTRAMOS LO QUE BUSCAMOS. Es importante esta situación, a veces encontramos y otras no.

Por lo anterior debemos pensar ¿cuándo termina el algoritmo?, lo que nos lleva a preguntarnos ¿termina en el mismo momento si lo buscado no está? La respuesta es que no es lo mismo. En caso de encontrar el elemento buscado, **el ciclo de búsqueda** termina antes del final del vector.

¿cuándo termina el algoritmo si no está el valor buscado?, vean que cambiamos la pregunta, dado que ya sabemos que en algunas ocasiones el valor buscado no está en el vector. En estos casos **el ciclo de búsqueda** termina antes al llegar al final del vector.

¿se tarda mucho en buscar si la lista es muy grande?, es una pregunta un tanto compleja, primero hay que definir qué es ser muy grande. Entonces tratemos de cambiar la pregunta: **¿ser tarda igual en los casos que el valor exista?,** la respuesta es NO. En los casos que existe el valor no se llega al final del vector en la búsqueda mientras que en aquellos casos que no existe se debe llegar al final (si el vector está desordenado) o se corta antes cuando el vector está ordenado.

Entonces se podría decir que en caso que i casi todos los valores buscados están en el vector, es indistinto que esté ordenado, mientras que en los casos que se busquen muchos valores que no existen, el algoritmo es más eficiente si busca en un vector ordenado ya que no debe recorrer hasta el final del vector.

Para vectores ordenados de N elementos, la cantidad de comparaciones tiende a $(N+1) / 2$ **PARECE UNA MEJORA AUNQUE NO MUY SIGNIFICATIVA**

El tema del tiempo de proceso en las búsquedas es importante debido a que se busca en una cantidad muy elevada de datos. Entonces hay algoritmos que consideran esto y logran una mejor performance, esto es buscan más rápido. El algoritmo de búsqueda binaria mejora muchísimo la búsqueda.

¿Entonces, hay varios algoritmos para realizar la búsqueda?,

Claro, podríamos decir que hay tantos algoritmos como desarrolladores, pero para se un poco más modestos veamos los siguientes:

BÚSQUEDA SECUENCIAL: se trata de recorrer todo el vector buscando el dato requerido. Finaliza cuando lo encuentra o cuando llega al final. Este proceso se basa en que el vector NO está ordenado.

BÚSQUEDA SECUENCIAL CON CENTINELA: el objetivo de esta búsqueda es que siempre se encuentre el dato buscado, para lo cual se agrega el dato buscado al final del vector. Luego se recorre el vector buscando el dato requerido, al cual siempre se lo va a encontrar: antes del final (el dato existe) o al final (el dato no existe).

BÚSQUEDA SECUENCIAL EN UN VECTOR ORDENADO: se trata de recorrer todo el vector buscando el dato requerido. Finaliza cuando lo encuentra o cuando el valor buscado es mayor al de la celda analizada. Se aplica solo cuando el vector está ordenado. En este método no se recorre todo el vector, salvo que el valor buscado esté (o deba estar) al final. **Es más eficiente que los anteriores**

BÚSQUEDA BINARIA: **este método es muy eficiente**, se utiliza solamente cuando el vector está ordenado. Se basa en "*dividir*" el vector por la mitad y evaluar si el dato buscado es mayor o menor al valor del medio. Este permite –con una única pregunta- descartar la mitad de los datos. Luego en la mitad en que probablemente esté el dato se aplica el mismo método, se "*divide*" a la mitad se evalúa ese valor y se sigue descartando. Entonces por cada pregunta se descarta la mitad de los valores. En muy pocas *iteraciones* se detecta si el valor está. En este caso la cantidad de comparaciones para un vector de N elementos es menor o igual a $\log_2(N/2)$

EN LAS PRÓXIMAS PÁGINAS SE DESARROLLAN CADA UNO DE LOS MÉTODOS.

Para ampliar propongo los siguientes artículos

" Diseño de un animador de algoritmos de búsqueda y ordenación ", María Luisa Pérez Delgado https://pdfs.semanticscholar.org/5616/e94e00d984ecb066990295567969a9fab87b.pdf

Un artículo mucho más abarcativo " Algoritmos de Búsqueda y Ordenamiento", M. L. Perez , P. A. Hernandez ´, J. Escuadra y M. P. Rubio https://www.inf.utfsm.cl/~noell/IWI-131-p1/Tema8b.pdf

Función de búsqueda en vectores desordenados

```
#include <iostream>
using namespace std;
#define CANT 10
typedef int tVector[CANT];
int buscarV(tVector, int, int);
void cargarVector(tVector, int);
void mostrarVector(tVector, int);
int main() {
  int i;
   tVector vect1;
   setlocale(LC_CTYPE, "Spanish");
  cargarVector(vect1,CANT);
   mostrarVector (vect1,CANT);
   i = buscarV(vect1,4,CANT);
   if ((i>-1))       cout << "se encontró el valor =
4" << endl;
    else   cout << "NO se encontró el valor = 4 "
<<endl;
    i = buscarV(vect1,13,CANT);
   if ((i>-1)) cout << "se encontró el valor = 13"
<< endl;
    else  cout << "NO se encontró el valor = 13"
<< endl;
  return 0;}
```

```
int buscarV(tVector vect1, int dato, int kTOPE) {
  int i = 0;
  while (((i<kTOPE) && (dato!=vect1[i])))
    i = i+1;
  if (((i>kTOPE) || (vect1[i-1]!=dato)))
    i = -1;
  return i;
}
```

Comparar por distinto (!=) cuando está desordenado

La función devuelve la posición del valor o -1 si no está

```
void cargarVector(tVector vect1, int kTOPE) {
  cout<<"datos del vector"<<endl;
  for (int i=0;i<kTOPE;i++)
     vect1[i] = 2*i;
}
```

cargarVector asignó los números pares consecutivos 2 a 20.

```
void mostrarVector(tVector vect1, int kTOPE) {
  for (int i=0;i<kTOPE;i++)
    cout<<vect1[i] << " - ";
  cout<<endl;
}
```

```
datos del vector
2 - 4 - 6 - 8 - 10 - 12 - 14 - 16 - 18 - 20

NO se encontró el valor = 4
NO se encontró el valor = 13

--------------------------------
```

Modifique el ejercicio para procesar datos y buscar valores pedidos desde el teclado

Función de búsqueda en vectores ORDENADOS

```
#include <iostream>
using namespace std;
#define CANT 10
typedef int tVector[CANT];
int buscarV(tVector, int, int);
void cargarVector(tVector, int);
void mostrarVector(tVector, int);
int main() {
  int i;
   tVector vect1;
   setlocale(LC_CTYPE, "Spanish");
  cargarVector(vect1,CANT);
   mostrarVector (vect1,CANT);
   i = buscarV(vect1,4,CANT);
   if ((i>-1))       cout << "se encontró el valor =
4" << endl;
    else   cout << "NO se encontró el valor = 4 "
<<endl;
    i = buscarV(vect1,13,CANT);
   if ((i>-1)) cout << "se encontró el valor = 13"
<< endl;
    else  cout << "NO se encontró el valor = 13"
<< endl;
  return 0;}
```

```
int buscarV(tVector vect1, int dato, int kTOPE) {
  int i = 0;
  while (((i<kTOPE) && (dato>=vect1[i])))
    i = i+1;
  if (((i>kTOPE) || (vect1[i-1]!=dato)))
    i = -1;
  return i;
}
```

Comparar por mayor o igual (>=) cuando está ordenado

La función devuelve la posición del valor o -1 si no está

```
void cargarVector(tVector vect1, int kTOPE) {
  cout<<"datos del vector"<<endl;
  for (int i=0;i<kTOPE;i++)
     vect1[i] = 2*i;
}
```

cargarVector asignó los números pares consecutivos 2 a 20.

```
void mostrarVector(tVector vect1, int kTOPE) {
  for (int i=0;i<kTOPE;i++)
    cout<<vect1[i] << " - ";
  cout<<endl;
}
```

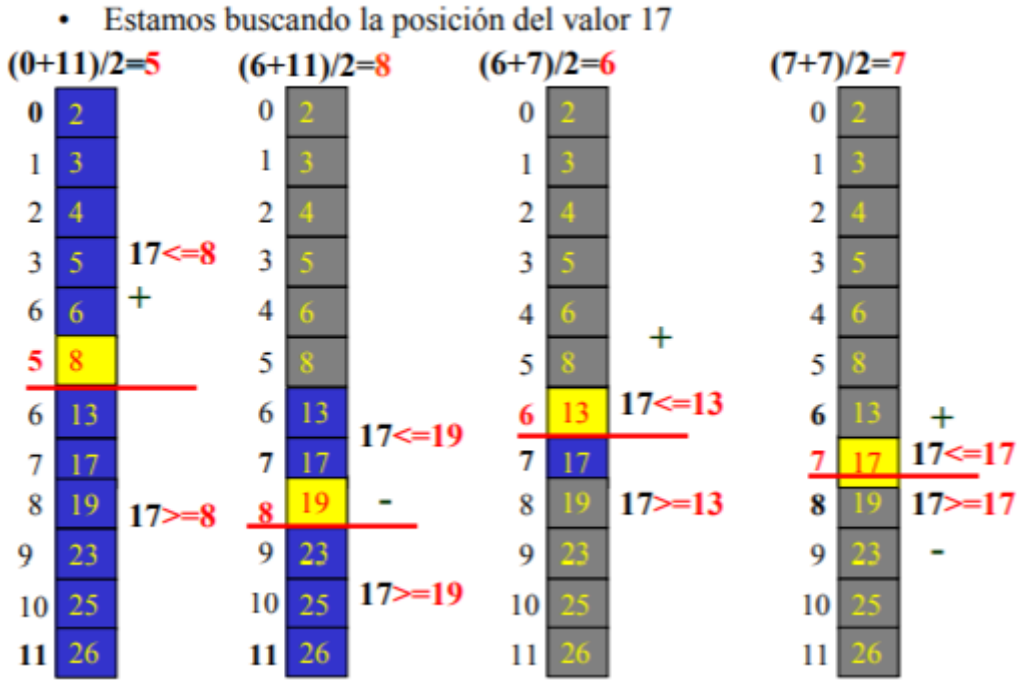
```
datos del vector
2 - 4 - 6 - 8 - 10 - 12 - 14 - 16 - 18 - 20

NO se encontró el valor = 4
NO se encontró el valor = 13

--------------------------------
```

Modifique el ejercicio para asignar números pedidos desde el teclado

Función de búsqueda binaria en vectores ORDENADOS

Mediante este criterio de búsqueda en **vectores ordenados**, se logra una mayor eficiencia dado que se intenta **no consultar** todos los valores secuencialmente, sino hacer el proceso de a "saltos".

Tal como buscamos en un conjunto de palabras ordenadas, primero evaluamos si el valor buscado está "antes" de la mitad, o "después" de la mitad. Así descartamos con una sola pregunta la mitad de los datos. Luego se repite el algoritmo en la mitad donde "podría" estar el valor buscado, y así sucesivamente hasta encontrar el valor o determinar que no está.



Las imágenes fueron tomadas del sitio Departamento de Informática Universidad Técnica Federico Santa María, Algoritmos de Búsqueda y Ordenamiento Prof.: Teddy Alfaro Olave. (https://www.inf.utfsm.cl/~noell/IWI-131-p1/Tema8b.pdf) . Desarrolla los algoritmos de búsqueda y ordenamiento y compara su eficiencia.

Ejemplo búsqueda binaria, en 1.100 valores

```
int buscarBinariaV(tVector v, int d, int kTOPE) {
    int primero=0;
    int ultimo=kTOPE-1;
    int ubi =  -1 ;
    int medio=(primero+ultimo)/2;
    while ((ubi==-1) && (primero<=ultimo)) {
    if (v[medio] < d)
          primero=medio+1;
       else if (v[medio] == d)
          ubi = medio ;
       else
          ultimo = medio - 1;
       medio = (primero+ultimo)/2;
    }
    if (primero>ultimo)
      ubi = -1;
    return ubi;
}
```

```
Busqueda binaria
================
datos del vector
primero=0 ultimo=1099 ubi=-1
primero=0 ultimo=548 ubi=-1
primero=0 ultimo=273 ubi=-1
primero=0 ultimo=135 ubi=-1
primero=0 ultimo=66 ubi=-1
primero=0 ultimo=32 ubi=-1
primero=0 ultimo=15 ubi=-1
primero=0 ultimo=6 ubi=-1
primero=0 ultimo=2 ubi=-1
primero=2 ultimo=2 ubi=-1
se encontró el valor = 4
```

```
primero=0 ultimo=1099 ubi=-1
primero=0 ultimo=548 ubi=-1
primero=0 ultimo=273 ubi=-1
primero=0 ultimo=135 ubi=-1
primero=0 ultimo=66 ubi=-1
primero=0 ultimo=32 ubi=-1
primero=0 ultimo=15 ubi=-1
primero=0 ultimo=6 ubi=-1
primero=4 ultimo=6 ubi=-1
primero=6 ultimo=6 ubi=-1
NO se encontró el valor = 13
```

Presten atención a los ejemplos: para 1.000 datos necesitó solamente 10 preguntas para determinar si el valor existe

Descripción del algoritmo de búsqueda:

"parte" el vector a la mitad;

compara el dato buscado con el valor del "medio¨;

en caso que no sea cambia los límites de búsqueda (primero y último)

repite hasta encontrar o hasta que el intervalo "desaparezca"

Funciones de ordenamiento

Ordenamiento de vectores

Existen diversos métodos de ordenar vectores, entre ellos los llamados selección, burbujeo, quick y shellsort.

El método de inserción consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño llevarlo a la segunda posición, y así sucesivamente hasta ordenarlo todo.

El método de burbujeo consiste en comparar elementos contiguos y en caso que estén desordenados, cambiarlos entre sí. Así, se compara el primero con el segundo, este con el tercero, el tercero con el cuatro y sucesivamente hasta llegar al final. Al cabo de todas comparaciones y cambios, se consigue que el mayor valor llegue al final. Se repite el proceso entre el primero y el anteúltimo, luego de ubicar al segundo mayor, se realiza el mismo proceso hasta llegar a tener en su lugar a los valores desde el tercero en adelante. Finalmente se compara el primero con el segundo y se ordena.

El método de selección simple consiste en buscar el menor, intercambiarlo con el primero, luego buscar entre el segundo y el último lugar el menor e intercambiarlo con el segundo y así sucesivamente hasta tener ordenado el vector.

El método Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

El método Quick sort mejora todo los ordenamientos anteriores, está basado en la técnica "divide y vencerás", es un algortimo "recursivo" y es el más rápido de todos los métodos de ordenamiento.

Finalmente existen métodos muy sofisticados y para un número muy grande de valores a ordenar.
Una curiosidad: hay métodos muy eficientes para pocos valores, que son ineficientes para un volumen muy grande de valores, y los hay muy ineficientes para un número pequeño, pero muy eficiente para un número muy grande de elementos.

Complejidad y eficiencia de los métodos de ordenamiento

La eficiencia se mide en cantidad de comparaciones e intercambios, dado que ambos procesos son "costosos". En la siguiente tabla vemos las complejidades para un vector de N elementos.

Método	Comparaciones
Burbuja	$N * (N + 1) / 2$
Selección	$N * (N - 1) / 2$
Inserción	$N * (N - 1) / 4$
Shell	N^2
Quick	$N * \log_2 (N)$

Ver algunos cálculos http://www.frlp.utn.edu.ar/materias/algoritmos/GUIACOMPLEJIDADDEALGORITMOS.pdf

Método de ordenamiento Inserción

```
#include<iostream>
using namespace std;
#define _TOPE 30
typedef int tVector[_TOPE];
void  pedirDatos(tVector vector, int & cant) ;
void  ordenar(tVector vector, int cant) ;
void  mostrar(tVector vector, int cant, string
cartel) ;
int main(){
  int cantVector = _TOPE;
  tVector vector;
  pedirDatos(vector, cantVector) ;
  mostrar(vector, cantVector, "antes") ;
  ordenar(vector, cantVector) ;
  mostrar(vector, cantVector,"despues") ;
  return 0;
 }
```

```
void  pedirDatos(tVector vector, int & cantVector) {
  int i=-1;
  int nro;
  cout<<"ingrese un numero (0=fin): "<<endl;
  cin>>nro;
  while ((i<cantVector)&&(nro!=0)) {
    i++ ;
    vector[i] = nro;
    cout<<"ingrese un numero (0=fin): "<<endl;
    cin>>nro;
  }
  cantVector = i+1 ;
};
```

```
void  mostrar(tVector vector, int cantVector, string
cartel) {
  cout<<cartel<<" vector=";
  for (int i=0;i<cantVector;i++){
     cout<<vector[i]<<" - ";
  }
  cout<<endl;
};
```

```
void  ordenar(tVector vector, int
cantVector) {
  int aux;
  int minimo;
  for(int i=0;i<cantVector;i++) {
     minimo = i ;
     for(int j=i+1;j<cantVector;j++)
        if(vector[j]<vector[minimo])
     minimo = j ;
     aux=vector[i];
     vector[i]=vector[minimo];
     vector[minimo]=aux;
   };
}
```

```
ingrese un numero (0=fin):
6
ingrese un numero (0=fin):
4
ingrese un numero (0=fin):
7
ingrese un numero (0=fin):
8
ingrese un numero (0=fin):
2
ingrese un numero (0=fin):
3
ingrese un numero (0=fin):
0
antes vector=6 - 4 - 7 - 8 - 2 - 3 -
despues vector=2 - 3 - 4 - 6 - 7 - 8 -
```

En este método se busca el menor y se lo intercambia con el primero, luego se busca el siguiente menor y se lo intercambia con el segundo y así sucesivamente.

Prestar atención a la cantidad de comparaciones que se realizan. Por cada recorrido del vector se ubica solamente un elemento en su ubicación ordenada.

Método de ordenamiento Burbuja

```
#include<iostream>
using namespace std;
#define _TOPE 30
typedef int tVector[_TOPE];
void  pedirDatos(tVector vector, int & cant) ;
void  ordenar(tVector vector, int cant) ;
void  mostrar(tVector vector, int cant, string
cartel) ;
int main(){
  int cantVector = _TOPE;
  tVector vector;
  pedirDatos(vector, cantVector) ;
  mostrar(vector, cantVector, "antes") ;
  ordenar(vector, cantVector) ;
  mostrar(vector, cantVector,"despues") ;
  return 0;
 }
```

```
void  pedirDatos(tVector vector, int & cantVector) {
  int i=-1;
  int nro;
  cout<<"ingrese un numero (0=fin): "<<endl;
  cin>>nro;
  while ((i<cantVector)&&(nro!=0)) {
    i++ ;
    vector[i] = nro;
    cout<<"ingrese un numero (0=fin): "<<endl;
    cin>>nro;
  }
  cantVector = i+1 ;
};
```

```
void  mostrar(tVector vector, int cantVector, string
cartel) {
  cout<<cartel<<" vector=";
  for (int i=0;i<cantVector;i++){
     cout<<vector[i]<<" - ";
  }
  cout<<endl;
};
```

```
void  ordenar(tVector vector, int
cantVector) {
  int aux;
  for(int i = 0; i < cantVector ; i++){
   for(int j = i + 1 ; j < cantVector ; j++){
     if(vector[i] > vector[j]) {
       aux=vector[i];
       vector[i]=vector[j];
       vector[j]=aux;
     }
   }
  }
}
```

```
ingrese un numero (0=fin):
6
ingrese un numero (0=fin):
4
ingrese un numero (0=fin):
7
ingrese un numero (0=fin):
8
ingrese un numero (0=fin):
2
ingrese un numero (0=fin):
3
ingrese un numero (0=fin):
0
antes vector=6 - 4 - 7 - 8 - 2 - 3 -
despues vector=2 - 3 - 4 - 6 - 7 - 8 -
```

En este método se compara el primer elemento con cada uno del resto y si están desordenados se los intercambia. Así se logra por cada recorrido completo del vector "enviar" el valor más chico hacia el inicio.

Prestar atención a la cantidad de comparaciones e intercambios que se realizan. Si el elemento mayor está al inicio, este recorrerá todas las celdas del vector hasta ubicarse.

Método de ordenamiento Selección

```
#include<iostream>
using namespace std;
#define _TOPE 30
typedef int tVector[_TOPE];
void  pedirDatos(tVector vector, int & cant) ;
void  ordenar(tVector vector, int cant) ;
void  mostrar(tVector vector, int cant, string
cartel) ;
int main(){
  int cantVector = _TOPE;
  tVector vector;
  pedirDatos(vector, cantVector) ;
  mostrar(vector, cantVector, "antes") ;
  ordenar(vector, cantVector) ;
  mostrar(vector, cantVector,"despues") ;
  return 0;
 }
```

```
void  pedirDatos(tVector vector, int & cantVector) {
  int i=-1;
  int nro;
  cout<<"ingrese un numero (0=fin): "<<endl;
  cin>>nro;
  while ((i<cantVector)&&(nro!=0)) {
    i++ ;
    vector[i] = nro;
    cout<<"ingrese un numero (0=fin): "<<endl;
    cin>>nro;
  }
  cantVector = i+1 ;
};
```

```
void  mostrar(tVector vector, int cantVector, string
cartel) {
  cout<<cartel<<" vector=";
  for (int i=0;i<cantVector;i++){
     cout<<vector[i]<<" - ";
  }
  cout<<endl;
};
```

```
void  ordenar(tVector vector, int
cantVector) {
  int aux;
  for(int i=0;i<cantVector;i++) {
    for(int j=0;j<cantVector-1;j++) {
      if(vector[j]>vector[j+1])        {
        aux=vector[j];
        vector[j]=vector[j+1];
        vector[j+1]=aux;
      }
   }
};
}
```

```
ingrese un numero (0=fin):
6
ingrese un numero (0=fin):
4
ingrese un numero (0=fin):
7
ingrese un numero (0=fin):
8
ingrese un numero (0=fin):
2
ingrese un numero (0=fin):
3
ingrese un numero (0=fin):
0
antes vector=6 - 4 - 7 - 8 - 2 - 3 -
despues vector=2 - 3 - 4 - 6 - 7 - 8 -
```

En este método se comparan elementos contiguos y si están desordenados se los intercambia. Así se logra por cada recorrido completo del vector "enviar" el valor más grande hacia el final.

Prestar atención a la cantidad de comparaciones e intercambios que se realizan. Si el elemento mayor está al inicio, este recorrerá todas las celdas del vector hasta ubicarse.

Método de ordenamiento Shell

```
#include<iostream>
using namespace std;
#define _TOPE 30
typedef int tVector[_TOPE];
void  pedirDatos(tVector vector, int & cant) ;
void  ordenar(tVector vector, int cant) ;
void  mostrar(tVector vector, int cant, string
cartel) ;
int main(){
  int cantVector = _TOPE;
  tVector vector;
  pedirDatos(vector, cantVector) ;
  mostrar(vector, cantVector, "antes") ;
  ordenar(vector, cantVector) ;
  mostrar(vector, cantVector,"despues") ;
  return 0;
 }
```

```
void  pedirDatos(tVector vector, int & cantVector) {
  int i=-1;
  int nro;
  cout<<"ingrese un numero (0=fin): "<<endl;
  cin>>nro;
  while ((i<cantVector)&&(nro!=0)) {
    i++ ;
    vector[i] = nro;
    cout<<"ingrese un numero (0=fin): "<<endl;
    cin>>nro;
  }
  cantVector = i+1 ;
};
```

```
void  mostrar(tVector vector, int cantVector, string
cartel) {
  cout<<cartel<<" vector=";
  for (int i=0;i<cantVector;i++){
     cout<<vector[i]<<" - ";
  }
  cout<<endl;
};
```

```
void ordenar(tVector vector, int cantVector)
{
 int aux, i , j , k , salto ;
 bool fin ;
   salto = cantVector ;
   while ( salto > 0 )  {
     salto = salto / 2 ;
     do  {
       fin = true ;
        k  = cantVector - salto ;
        for ( i = 0 ; i < k ; i++ ) {
          j = i + salto ;
          if ( vector [ i ] > vector [ j ]) {
            aux = vector [ i ] ;
            vector [ i ] = vector [ j ] ;
            vector [ j ] = aux ;
            fin = false ;
          }
        }
      }
      while ( ! fin ) ;
    }
}
```

```
ingrese un numero (0=fin):
6
ingrese un numero (0=fin):
4
ingrese un numero (0=fin):
7
ingrese un numero (0=fin):
8
ingrese un numero (0=fin):
2
ingrese un numero (0=fin):
3
ingrese un numero (0=fin):
0
antes vector=6 - 4 - 7 - 8 - 2 - 3 -
despues vector=2 - 3 - 4 - 6 - 7 - 8 -
```

Este método va "particionando el vector de a mitades y compara valores "alejados". Esto es el primero con el del medio, el segundo con el siguiente y así hasta el final. Luego parte el intervalo y compara el primero con el que está a la mitad de la mitad. Y así sucdesivamente.

Método de ordenamiento Quick

```
#include<iostream>
using namespace std;
#define _TOPE 30
typedef int tVector[_TOPE];
void  pedirDatos(tVector vector, int & cant) ;
void  ordenar(tVector vector, int desde, int hasta) ;
void  mostrar(tVector vector, int cant, string cartel)
;
int main(){
  int cantVector = _TOPE;
  tVector vector;
  pedirDatos(vector, cantVector) ;
  mostrar(vector, cantVector, "antes") ;
  ordenar(vector, 1, cantVector) ;
  mostrar(vector, cantVector,"despues") ;
  return 0;
 }
```

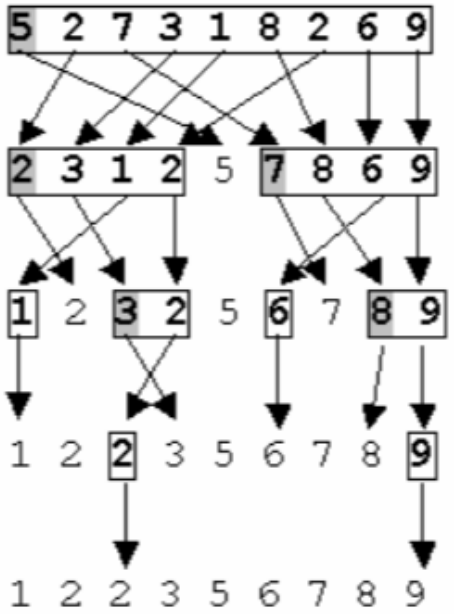
```
void ordenar(tVector vector, int desde, int hasta){
 int pivote;
 if(desde < hasta){
   pivote=colocar(vector, desde, hasta);
   ordenar(vector, desde, pivote-1);
   ordenar(vector, pivote+1, hasta);
 }
}
```

```
int colocar(tVector vector, int desde, int
hasta){
 int i;
 int pivote, valor_pivote;
 int temp;
 pivote = desde;
 valor_pivote = vector[pivote];
 for (i=desde+1; i<=hasta; i++){
  if (vector[i] < valor_pivote){
    pivote++;
    temp=vector[i];
    vector[i]=vector[pivote];
    vector[pivote]=temp;
  }
 }
 temp=vector[desde];
 vector[desde]=vector[pivote];
 vector[pivote]=temp;
 return pivote;
}
```

```
ingrese un numero (0=fin):
6
ingrese un numero (0=fin):
4
ingrese un numero (0=fin):
7
ingrese un numero (0=fin):
8
ingrese un numero (0=fin):
2
ingrese un numero (0=fin):
3
ingrese un numero (0=fin):
0
antes vector=6 - 4 - 7 - 8 - 2 - 3 -
despues vector=2 - 3 - 4 - 6 - 7 - 8 -
```

Este método es muy sofisticado, utiliza técnicas recursivas para su resolución (observen que la función ordenar se invoca a si misma.

Es muy eficiente para un número muy grande de datos.



Para ampliar

En la etiqueta RECURSOS OPTATIVOS van a encontrar animaciones para cada uno de los métodos descriptos en este documento. Muestra los códigos de cada método en JAVA (verán que es un lenguaje de programación similar (al menos en esta etapa) al C++

Agradezco a la profesora María Luisa Pérez Delgado de la Escuela Politécnica Superior de Zamora, Universidad de Salamanca quien gentilmente me envió las animaciones publicadas por la Universidad Carlos III Madrid.

Los métodos son:

- Inserción
- Burbuja
- QuickSOrt
- Selección
- Shell

Para avanzar y profundizar se puede consultar método de burbujeo:

https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja
Método de inserción : https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n
Otros http://pseint.sourceforge.net/index.php?page=ejemplos.php&cual=OrdenaLista&mode=flexible
Shell; http://www.algofun.com/index.php/es/algoritmosycodigo/ordenamiento/Shell; https://www.ecured.cu/Algoritmo_de_Ordenamiento_Shell
Quick https://www.ecured.cu/QuickSort
Selección https://www.ecured.cu/Algoritmo_de_ordenamiento_por_selecci%C3%B3n
Diversos métodos http://biolab.uspceu.com/aotero/recursos/docencia/TEMA%208.pdf; https://www.cimat.mx/~alram/comp_algo/clase15.pdf

Ejercicio

Bajar el programa en "comparacionmetodos" C++ de la carpeta "Algunos ejemplos en C++".

Compilarlo

Ver los resultados

Cambiar las constantes TOPE y RANGOVALORES

Ver los resultados.

Este programa cuenta la cantidad de comparaciones de cada método y muestra el valor. Prestar atención cómo se van "alejando" la cantidad de comparaciones a medida que "crece" el vector.