Unidad 4 cont.

Clases de Almacenamiento de Variables

(Capitulo 6)

Algoritmos de Búsqueda y Ordenamiento

(Capitulo 11 bibliografía)

» Ing. Ventre, Luis O.

OBJETIVO

- El ALCANCE (local, global) de una variable, puede verse como la **dimension espacial** dentro de un programa donde esa variable es válida.

- Las variables también tienen una **dimensión temporal** dentro de la ejecución de un programa, y ésta puede alterarse como veremos a continuación.

- El espacio temporal, se refiere al tiempo de «vida» de la variable.

OBJETIVO

- Las variables, e identificadores **consumen recursos**. Algunas variables solo son utilizadas en pequeñas partes del código, por lo que no es necesario que existen a lo largo de todo el programa. En algunas es imprescindible que no pierdan su valor; e incluso a veces es necesario utilizar el mismo identificador para distintas variables.

- Todo estos problemas son resueltos a través de las **clases de almacenamiento de las variables**. Con ellas es posible modificar el alcance de las variables, y la **duración temporal** de reserva de los recursos.

INTRODUCCION

Variable:

Referencia semántica o **nombre** de un **espacio de memoria reservado** para almacenar un valor.

Toda variable tiene asociado:

- – Un **tipo de dato** (booleano, entero, carácter, decimal, puntero..etc)
- – Un **alcance** (dimensión física) (Global o Local)
- – *Una **clase de almacenamiento** (dimensión temporal)*

Clases y Utilización

Clases de almacenamiento de variables:

Auto	Static	Extern	Register
(Por omisión)	Persistencia/Privacidad	Alcance extendido	Optimización

Ej. de declaraciones de vbles. con clase de almacenamiento:

auto **int** numero
register **float** factor

static **int** millas
extern **int** watts

Clases y Utilización

Clases de almacenamiento de **variables locales**:

Auto Static Register

Clases de almacenamiento de **variables globales**:

Static Extern

Clases de Almac. Vbles **LOCALES**

La clase **AUTO**:

- Mas utilizada**. Por omisión**. Su nombre deriva de automática.
- Omitir la declaración de la clase en una función implica que la vble. es "*auto*".
- El almacenamiento es **reservado** cada vez que se llama a la función y durante su ejecución (viva), y **devueltos** cuando finaliza la ejecución de la misma (muerta).
- El proceso anterior se repite cada vez que es llamada la función.

Clases de Almac. Vbles **LOCALES**

Ejemplo de clase AUTO:

```
int main()
{
int cuenta;
for(cuenta=1;cuenta <=3; cuenta ++)
{
            probarclaseauto();
}
return 0;
}
```

La salida producida es:

El valor de la vble. Automatica es 0
El valor de la vble. Automatica es 0
El valor de la vble. Automatica es 0

```
void probarclaseauto()
{
auto int num=0;  // ..o solo “int num” ya que por omision se crea auto

cout<<“ El valor de la vble. Automatica es”<<num<<endl;
num++;
}
```

Clases de Almac. Vbles **LOCALES**

La clase STATIC:

- A veces es necesario **recordar el valor** por más que finalice la ejecución de la función.

- Una vez creadas, las vbles. **locales** static, permanecen con “vida” hasta la **finalización del programa**.

- Estas variables **no se inicializan** en tiempo de ejecución.

- Cuando no se da una inicialización explicita, **son inicializadas a 0**.

Clases de Almac. Vbles **LOCALES**

Ejemplo de clase STATIC:

```
int main()
{
int cuenta;
for(cuenta=1;cuenta <=3; cuenta ++)
{
        probarstatic();
}
return 0;
}
```

La salida producida es:

El valor de la vble. STATIC ahora es 0
El valor de la vble. STATIC ahora es 1
El valor de la vble. STATIC ahora es 2

```
void probarstatic()
{
static int num=0;  // ..o solo " static int num" ya que por omisión se inicializa 0

cout<<" El valor de la vble. STATIC ahora es"<<num<<endl;
num++;
}
```

Clases de Almac. Vbles **LOCALES**

Cual es la diferencia entre las siguientes funciones, que resultado dará el invocarlas 3 veces?

```
void func1()
{
static int años = 1;
cout<<“El valor de años es” << años<<endl;
años = años + 2;
}
```

La salida producida es:

El valor de años es 1
El valor de años es 3
El valor de años es 5 …

```
void func2()
{
static int años;
años = 1;
cout<<“El valor de años es” << años<<endl;
años = años + 2;
}
```

La salida producida es:

El valor de años es 1
El valor de años es 1
El valor de años es 1…
Impide el propósito de la vble.
Static!!!!!

Clases de Almac. Vbles **LOCALES**

Ejemplo: Declaración de variable

```
…
register double factor;
…
```

La clase REGISTER:

- Su utilización no es tan extensa.
- **Duración igual** que clase auto. Es decir se crea cuando se introduce la función que la declara y se destruye cuando se completa su ejecución.
- La única diferencia con la variable auto es **donde** se localiza el almacenamiento.
- Única restricción, imposibilidad de usar el operador de direccionamiento &. Puede ser ignorada por compilador.

Clases de Almac. Vbles **GLOBALES**

La clase EXTERN en variables globales:

- Una variable global, **existe** hasta la finalización del programa. Las clases de almacenamiento en estas variables **afectan el alcance** no la duración.

- Propósito **extender el alcance** mas allá de los limites normales.

- La declaración NO origina una variable nueva, y **no reserva un espacio nuevo** de almacenamiento.

- La **inicialización** va en **declaración original**, no en extern.

Clases de Almac. Vbles **GLOBALES**

Archivo 1

```
int distancia=100;
double metros=0;
static double factor;
extern int trabajo;
      .
      .
int main()
{
Func1();
Func2();
Func3();
Func4();
      .
}
```

Archivo 2

```
int trabajo=0;
extern double metros;

int func3()
{
     ….
}

int func4()
{
    extern int distancia;
      …
```

Extienden el alcance de las variables globales de cada archivo

Clases de Almac. Vbles **GLOBALES**

La clase STATIC en variables globales:

- Objetivo, **prevención de extensión** de una variable global a un segundo archivo.
- Declaración idéntica a vbles. locales static, pero difiere el lugar de la instrucción de declaración.
- Proporciona un **grado de privacidad**. Otros archivos no pueden acceder ni cambiar valores.
- Nunca una vble. **global static** puede extendida a otro archivo mediante el uso de extern.
- Por defecto una vble. global **puede ser** extendida a otra fte.

Integrando conceptos

Archivo 1

```
char elección;
int marcador=0;
float fecha, tiempo;

int main()
{  ….
}

int empuje()
{
double ultimo=0;
int const;
 …
}
```

Archivo 2

```
char tipo_b;
double resistencia;

double roi()
{   ...
}

double promedio()
{   …
double calculo=0;
}
```

static float fecha;

extern int marcador;

extern char elección;

static double calculo=0

Por omision es “auto” y no guarda su valor!...

register double ultimo=0

[illegible] invocaciones a la función?...

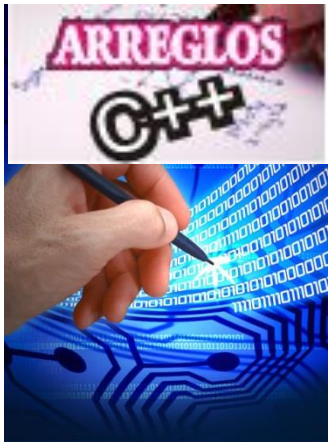
[illegible] registro de acceso rápido?...

Unidad 5

Arreglos cont.

Algoritmos de Búsqueda y Ordenamiento

(Capitulo 11 bibliografía)



» Ventre, Luis O.

ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

Hay, muchas ocasiones en donde el programador se enfrenta a la tarea de **buscar en una lista** si se encuentra un nombre, y/o tener que **ordenar una lista** de manera ascendente o de acuerdo al orden alfabético. Para hacerlo existen algoritmos de búsqueda y ordenamiento.

Entre los algoritmos de BUSQUEDA los dos métodos mas comunes de búsqueda son los siguientes:

- **BUSQUEDA LINEAL**
- **BUSQUEDA BINARIA**

BUSQUEDA LINEAL:

Es el método **mas simple**, también llamado “búsqueda secuencial”, cada elemento de la lista se compara con el objeto deseado hasta que hay una coincidencia o se termina la lista.

No es lo mas eficiente en listas alfabéticas largas.

Sin embargo tiene las siguientes ventajas

- **Algoritmo Simple.**
- **La lista no requiere estar en un orden particular**

Este algoritmo comienza con el primer elemento de la lista, y continua su análisis hasta el ultimo elemento o match.

ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

El pseudocódigo de este algoritmo será:

Para todos los elementos de la tabla
Compare el elemento con el elemento buscado
Si el elemento se encuentra
Devolver el valor de índice del elemento actual
Termina el si
Termina el Para
Devolver -1 si el elemento no se encontró.

```
int busquedaLineal(int lista[], int tamanho, int clave)
{
  int i;

  for (i = 0; i < tamanho; i++)
  {
    if (lista[i] == clave)
      return i;
  }

  return -1;
}
```

ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

```
#include <iostream>
using namespace std;


int busquedaLineal(int [], int, int)

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {5,10,22,32,45,6
  int item, ubicacion;


  cout << "Ingrese el item que esta
  cin  >> item;

  ubicacion = busquedaLineal(nums, NUMEL, item);

  if (ubicacion > -1)
    cout << "El item fue encontrado en la ubic
         << ubicacion << endl;
  else
    cout << "El item no fue encontrado en la lista\n";

  system("PAUSE");
  return 0;
}
```

```
// esta función retorna la ubicacion de la clave en la lista
// se devuelve un -1 si el valor no es encontrado
int busquedaLineal(int lista[], int tamanho, int clave)
{
  int i;

  for (i = 0; i < tamanho; i++)
  {
    if (lista[i] == clave)
      return i;
  }

  return -1;
}
```

```
Ingrese el item que esta buscando: 45
El item fue encontrado en la ubicacion 4
Presione una tecla para continuar . . .
```

BUSQUEDA BINARIA:

En este método la lista **DEBE** estar ordenada.

El elemento buscado se compara con el elemento del **"medio de la lista";** pudiendo existir 3 resultados:

- Que el número sea el buscado, lo cual indica el fin de la búsqueda,
- Que el número sea mayor que el resultado, por lo cual se descarta la primer mitad y se hace el mismo ciclo para la segunda mitad.
- Que el número sea menor y se hace lo análogo con la primera mitad.

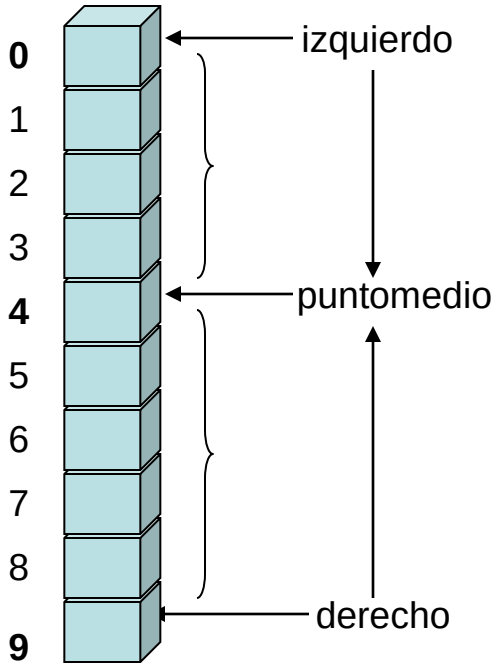
Ventaja:

El numero de elementos donde debe buscarse se va **reduciendo a la mitad con cada ciclo de comparación**.

En una lista de n elementos, luego de la primera búsqueda solo resta verificar n/2 elementos, ya que se han descartado n/2, y en el siguiente paso n/4 y así sucesivamente.

Pseudocódigo: VER en la pagina 656 BIBLIOGRAFIA PRINCIPAL

```
int busquedaBinaria(int lista[], int tamanho, int clave)
{
  int izquierdo, derecho, puntomedio;

  izquierdo = 0;
  derecho = tamanho -1;

  while (izquierdo <= derecho)
  {
    puntomedio = (int) ((izquierdo + derecho) / 2);
    if (clave == lista[puntomedio])
    {
      return puntomedio;
    }
    else if (clave > lista[puntomedio])
      izquierdo = puntomedio + 1;
    else
      derecho = puntomedio - 1;
  }

  return -1;
}
```



ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

```
Ingrese el item que esta buscando: 45
El item fue encontrado en la ubicacion 4
Presione una tecla para continuar . . .
```

```cpp
#include <iostream>
using namespace std;

int busquedaBinaria(int [], int, int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {5,10,22,32,45,67,73,98
  int item, ubicacion;

  cout << "Ingrese el item que esta buscand
  cin  >> item;
  ubicacion = busquedaBinaria(nums, NUMEL,
  if (ubicacion > -1)
    cout << "El item fue encontrado en la u
         << ubicacion << endl;
  else
    cout << "El item no fue encontrado en e

  system("PAUSE");
  return 0;
}
```

```cpp
// esta función retorna la ubicacion de la clave en la lista
// se devuelve un -1 si el valor no es encontrado
int busquedaBinaria(int lista[], int tamanho, int clave)
{
  int izquierdo, derecho, puntomedio;

  izquierdo = 0;
  derecho = tamanho -1;

  while (izquierdo <= derecho)
  {
    puntomedio = (int) ((izquierdo + derecho) / 2);
    if (clave == lista[puntomedio])
    {
      return puntomedio;
    }
    else if (clave > lista[puntomedio])
      izquierdo = puntomedio + 1;
    else
      derecho = puntomedio - 1;
  }

  return -1;
}
```

ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

En la pagina 660 puede observarse una comparación de recorridos o ciclos por cada método para encontrar el elemento, dependiendo de la cantidad de elementos de la lista.

Tamaño Arreglo	10 elem.	50 elem.	50.000 elem.	500.000 elem.
Recorridos de busqueda lineal promedio.	5	25	25.000	250.000
Recorridos Busq. Binaria maximos	4	6	16	19

De esta comparación se observa que un criterio adecuado seria para listas con menos de 50 elementos es aceptable el método de búsqueda lineal o secuencial, para listas mayores debe usarse búsqueda binaria.

ALGORITMOS DE ORDENAMIENTO

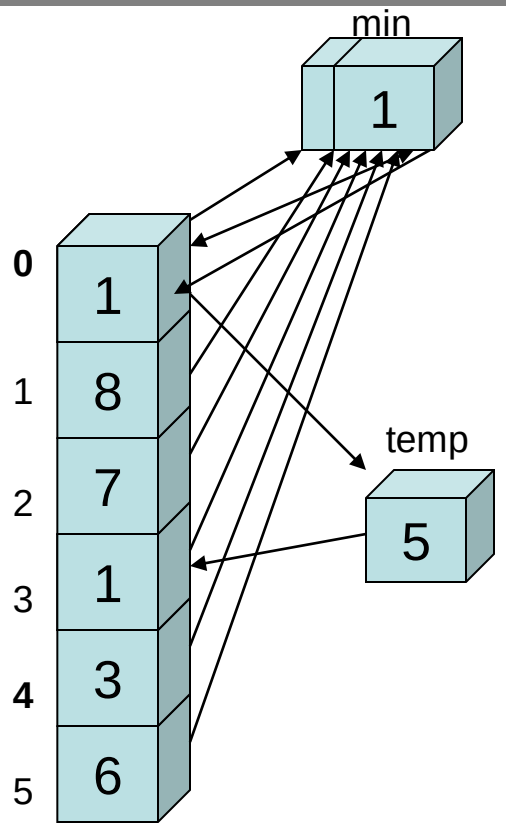
Existen 2 categorías de tecnicas de ordenamiento, **internos y externos;** lo cual hace referencia a si la lista puede ordenarse cargándose en la memoria de la computadora o no, lo que esta vinculado al tamaño de la lista a ordenar. Veremos 2 métodos de algoritmos de ordenamiento INTERNO.

ORDENAMIENTO POR SELECCIÓN:

Inicia con la selección **del valor mas pequeño de la lista completa y se intercambia con el primero**. Luego de esta selección e intercambio, el siguiente elemento mas pequeño en la lista restante se selecciona e intercambia con el segundo elemento. Este segundo recorrido va desde el segundo al ultimo. Esto se repite en una lista de n elementos n-1 veces.

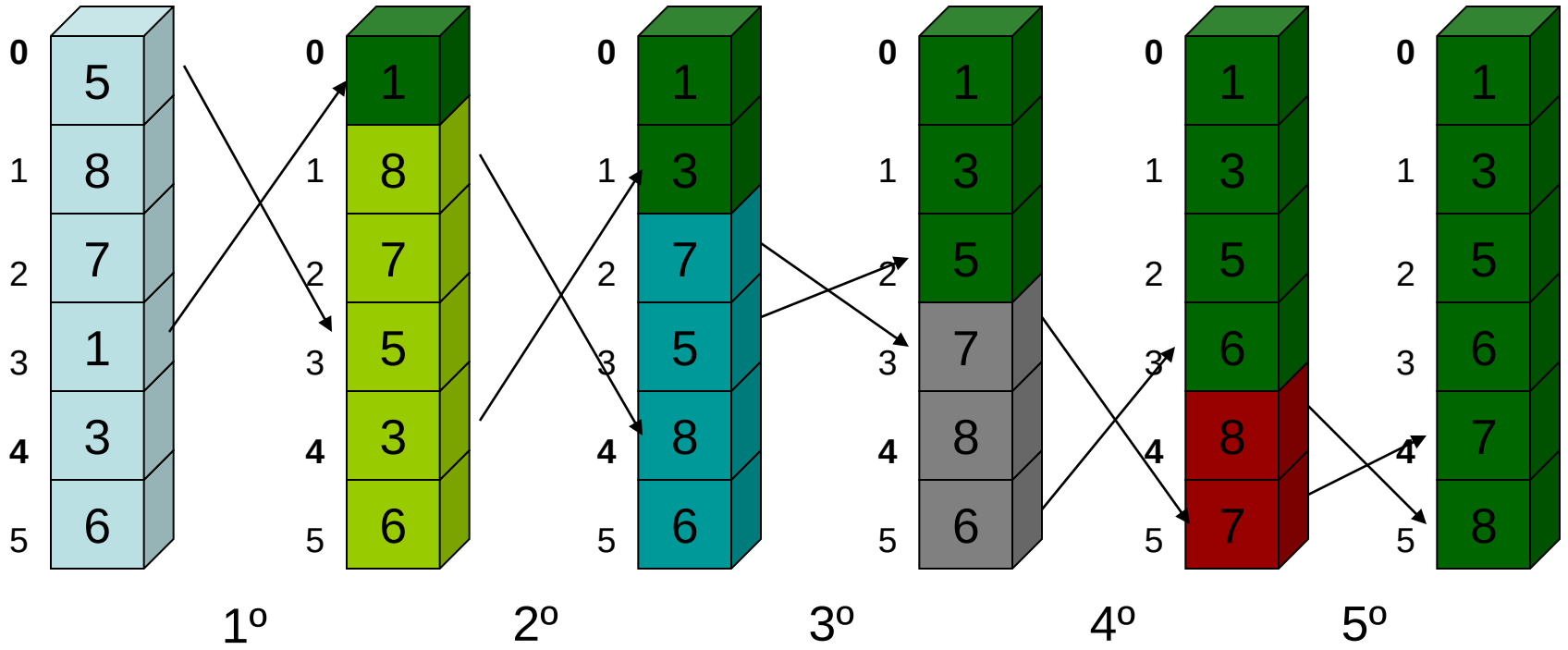
ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

```
int ordenSeleccion(int num[], int numel)
{
  int i, j, min, indicemin, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    min = num[i];    // supone que el minimo es el primer elemento
    indicemin = i;   // indice del elemento mínimo
    for(j = i + 1; j < numel; j++)
    {
      if (num[j] < min)    // si se ha localizado un valor inferior
      {                    // se lo captura
         min = num[j];
         indicemin = j;
      }
    }
    if (min < num[i])  // verifica si hay un nuevo mínimo
    {                  // en cuyo caso intercambia valores
      temp = num[i];
      num[i] = min;
      num[indicemin] = temp;
      movimientos++;
    }
  }

  return movimientos;
}
```



ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

Algoritmo de Ordenamiento por selección:



Iteración.

Arreglo con 6 elementos. N – 1 iteraciones.

ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

```
#include <iostream>
using namespace std;

int ordenSeleccion(int [], int);

int main()
{
  const int NUMEL = 6;
  int nums[NUMEL] = {5,8,7,1,3,6};
  int i, movimientos;

  movimientos = ordenSeleccion(nums,

  cout << "La lista ordenada en orden
  for (i = 0; i < NUMEL; i++)
    cout << "   " <<nums[i];

  cout << endl << "Se hicieron " <<
       << " movimientos para ordenar

  system("PAUSE");
  return 0;
}
```

```
La lista ordenada en orden ascendente, es:
  1   3   5   6   7   8
Se hicieron 5 movimientos para ordenar esta lista
Presione una tecla para continuar . . .
```

```
int ordenSeleccion(int num[], int numel)
{
  int i, j, min, indicemin, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    min = num[i];    // supone que el minimo es el primer elemento
    indicemin = i;   // indice del elemento mínimo
    for(j = i + 1; j < numel; j++)
    {
      if (num[j] < min)    // si se ha localizado un valor inferior
      {                    // se lo captura
         min = num[j];
         indicemin = j;
      }
    }
    if (min < num[i])  // verifica si hay un nuevo mínimo
    {                  // en cuyo caso intercambia valores
      temp = num[i];
      num[i] = min;
      num[indicemin] = temp;
      movimientos++;
    }
  }

  return movimientos;
}
```

ORDENAMIENTO DE LA BURBUJA - POR INTERCAMBIO:

Este método basa su funcionamiento en el intercambio de elementos adyacentes de tal manera que la lista quede ordenada.

Se **comparan** los **elementos sucesivos** de una lista comenzando por los dos primeros.

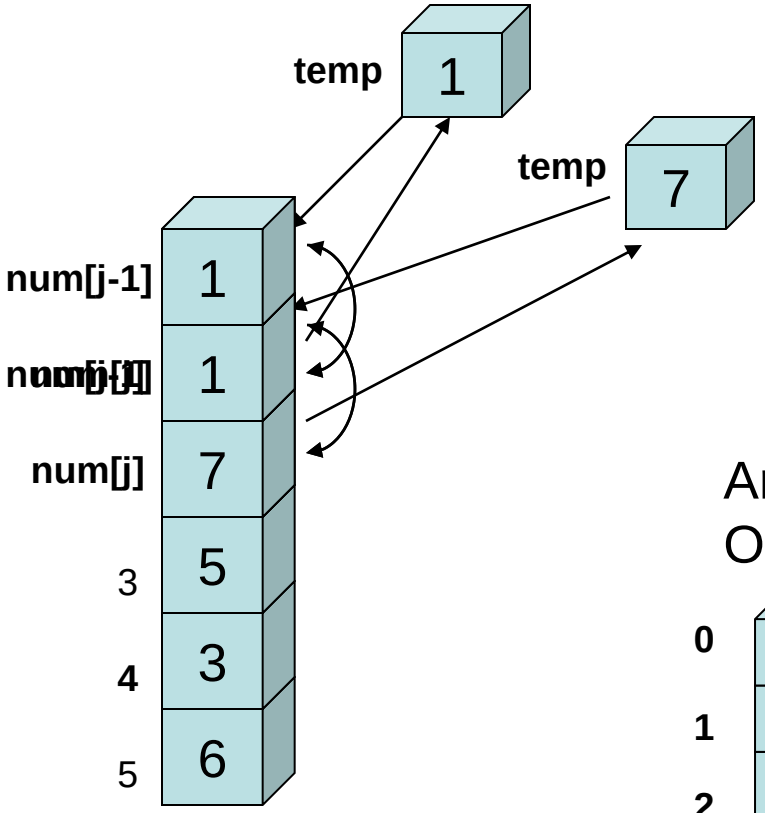
Dependiendo si el orden será ascendente o descendente se ubican los elementos.

Luego se compara el segundo con el tercero y así sucesivamente. Debemos observar que esto asegura que el **mayor elemento quede al ultimo**. Si hubo cambios en la secuencia debe ejecutarse nuevamente pero sin incluir el ultimo elemento.

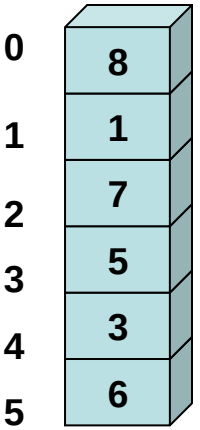
El ultimo no se cuenta, porque el método hunde al fondo de la lista el valor mas positivo.

ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

```
int ordenBurbuja(int num[], int numel)
{
  int i, j, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    for(j = 1; j < numel; j++)
    {
      if (num[j] < num[j-1])
      {
        temp = num[j];
        num[j] = num[j-1];
        num[j-1] = temp;
        movimientos++;
      }
    }
  }

  return movimientos;
}
```

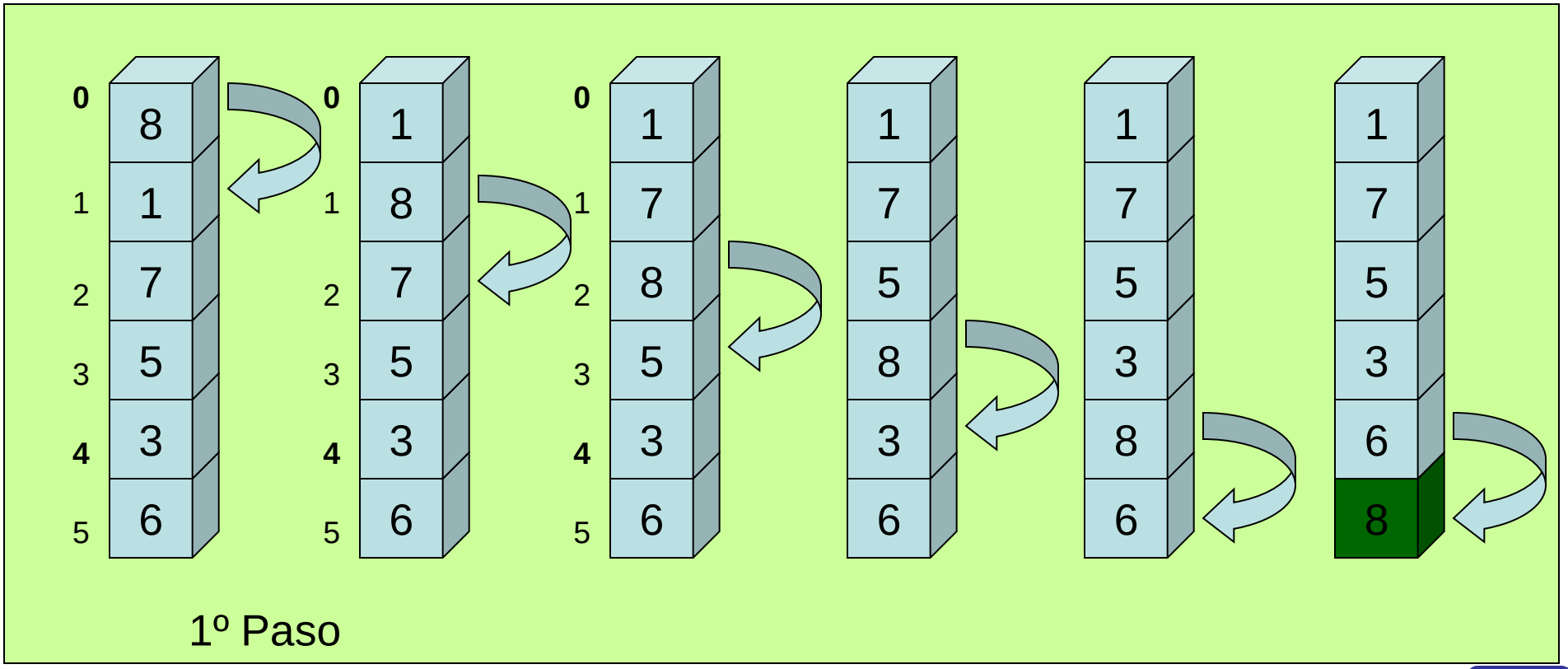


Arreglo Original



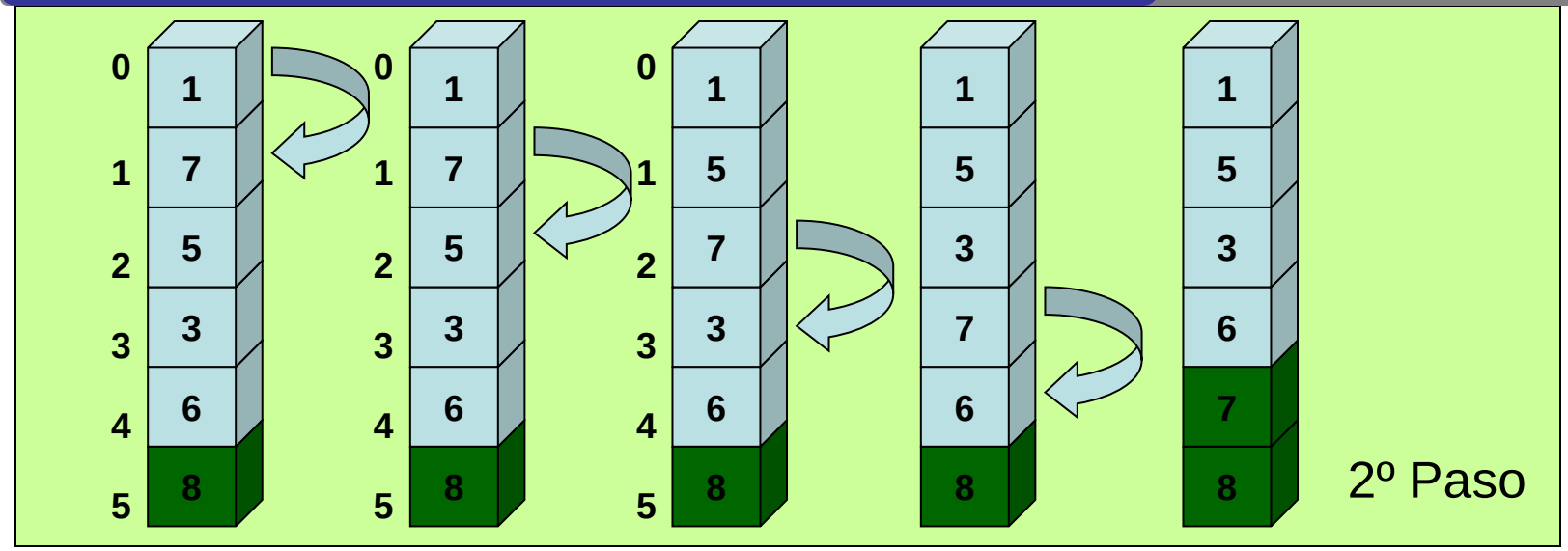
ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

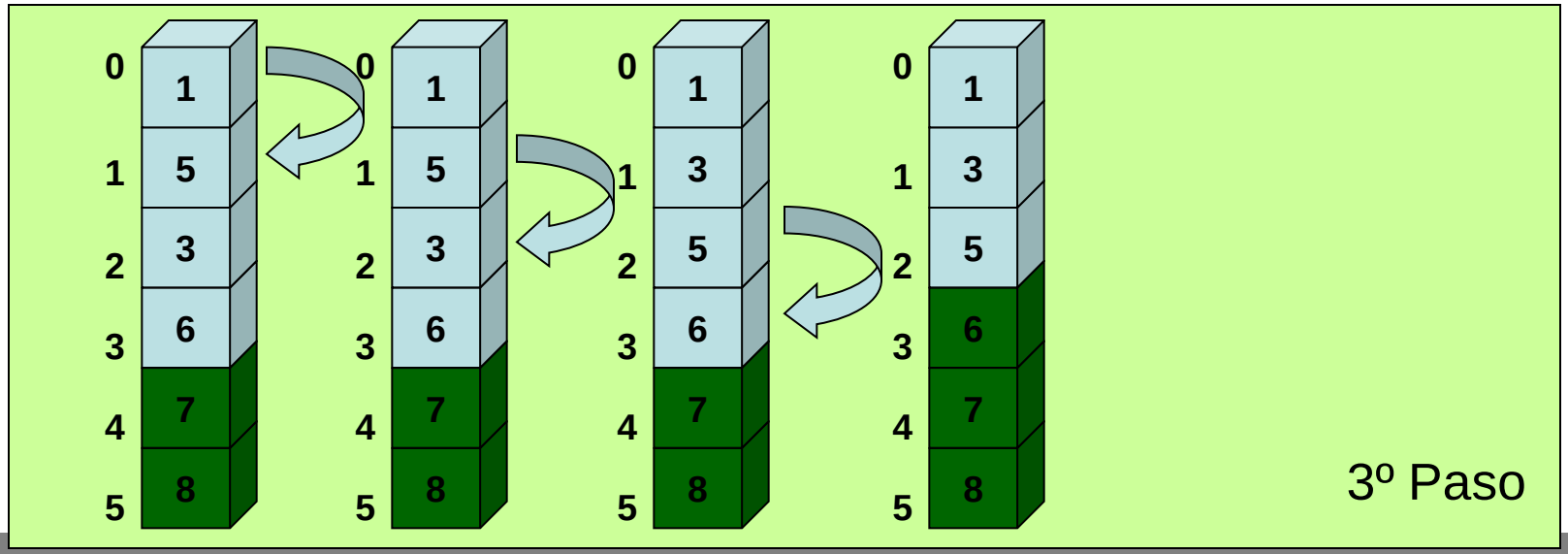
Por lo general el ordenamiento por selección responde igual o mejor que el ordenamiento de la burbuja. De todas maneras el orden de ordenamiento de ambos métodos es el mismo, pero en casos especiales se comporta mejor el método por selección.



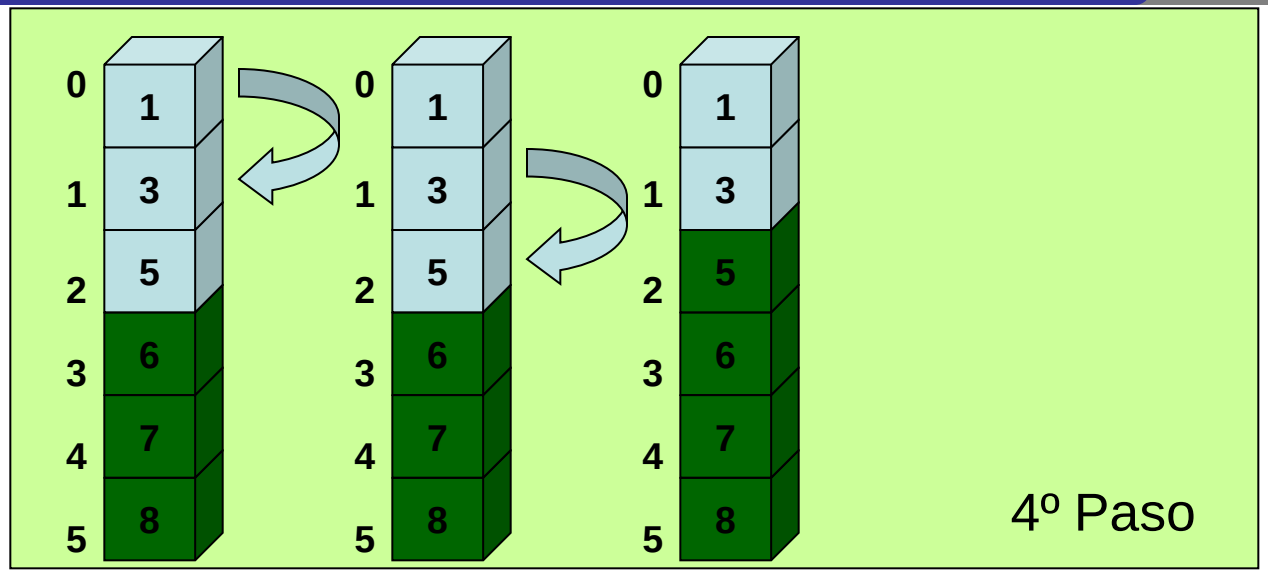
1º Paso

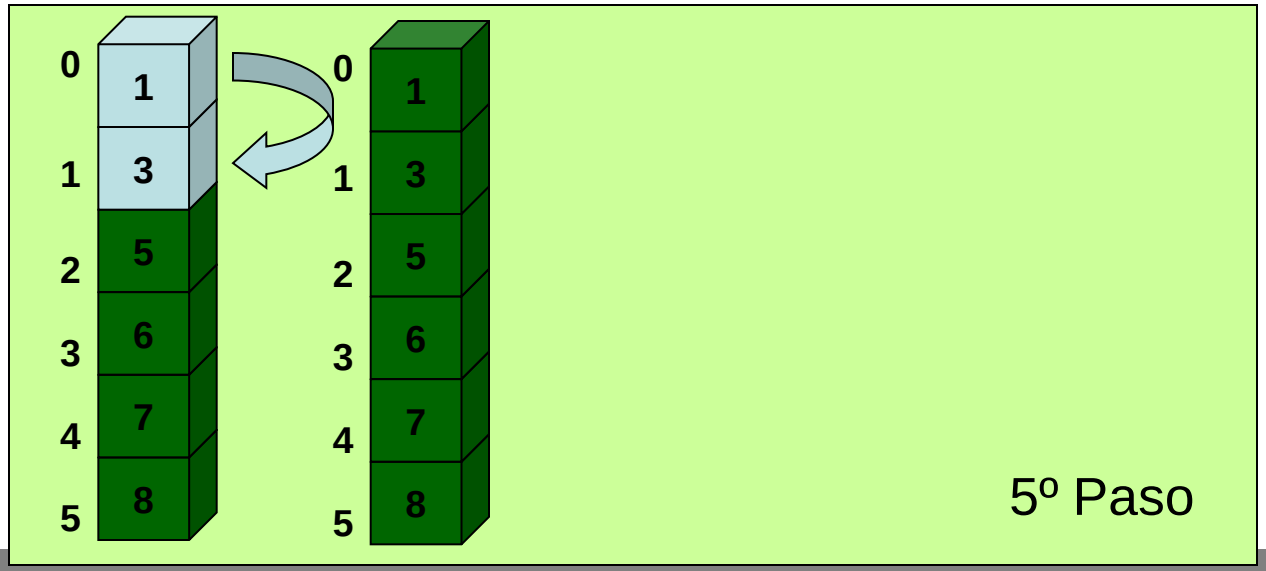
ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO





ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO





ALGORITMOS DE BUSQUEDA Y ORDENAMIENTO

```
#include <iostream>
using namespace std;

int ordenBurbuja(int [], int);

int main()
{
  const int NUMEL = 6;
  int nums[NUMEL] = {8,1,7,5,3,6};
  int i, movimientos;

  movimientos = ordenBurbuja(nums, NUMEL);

  cout << "La lista ordenada en orden ascendente,
  for (i = 0; i < NUMEL; ++i)
    cout << "  " <<nums[i];

  cout << endl << "Se hicieron " <<  movimientos
       << " movimientos para ordenar esta lista\n

  system("PAUSE");
  return 0;
}
```

```
La lista ordenada en orden ascendente, es:
  1  3  5  6  7  8
Se hicieron 9 movimientos para ordenar esta lista
Presione una tecla para continuar . . .
```

```
int ordenBurbuja(int num[], int numel)
{
  int i, j, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    for(j = 1; j < numel; j++)
    {
      if (num[j] < num[j-1])
      {
        temp = num[j];
        num[j] = num[j-1];
        num[j-1] = temp;
        movimientos++;
      }
    }
  }

  return movimientos;
}
```