Ordenación y búsqueda de datos

Ordenar datos es una tarea muy importante y frecuente en programación, además está relacionada directamente con el manejo de arreglos.

Para ilustrar esto, veamos el siguiente ejemplo: queremos elaborar una aplicación que permita para un supermercado, dada una lista de compras de un cliente, determinar el total de la compra. La solución de este problema va por tomar uno a uno los artículos buscar en una tabla de artículos el productos, determinar el costo y agregar este monto al total. Pues bien, las búsquedas que hagamos en el proceso puede ser una tarea que tome la mayor parte del tiempo en la ejecución del programa, por lo que esta tarea debe realizarse con mucha eficiencia.

Para entender esto veamos la siguiente analogía: pensemos en una guía telefónica, esto es un libro de miles de páginas en donde se registran todos los usuarios que poseen un teléfono. Si alguien quiere buscar el teléfono de una persona allí, a nadie se le ocurriría buscarlo desde la primera página, leyendo uno por uno los nombres que allí aparezcan. Lo que uno hace es abrir la guía más o menos por la mitad y suponiendo que en esas páginas encontramos un usuario que se apellide por ejemplo "Márquez", continuaremos la búsqueda dependiendo del nombre que buscamos. Por ejemplo, si buscamos a Rodríguez sabremos que lo encontraremos en la mitad de la derecha, pero si buscamos a Jiménez estará a la izquierda. Luego repetimos la operación con la mitad correspondiente hasta ubicarlo.

Esta manera tan rápida de encontrar un teléfono en la guía se puede hacer por una razón importante, los usuarios están ordenados por el nombre de las personas. Imagínese que pasaría si en la guía que tenemos, los nombres de las personas estuvieran colocados conforme fueron adquiriendo su teléfono, ¿Cuánto tiempo nos tomaría encontrar el número de teléfono de una persona?

Volviendo a la aplicación del supermercado, si los datos no estuvieran ordenados, por más rápida que puede ser la computadora, entregar el total de una lista de compras demoraría tanto que seguramente el supermercado perdería mucha clientela.

Como este ejemplo, la mayoría de aplicaciones a la que nos enfrentemos tendrán que realizar alguna búsqueda que, de manera general, deba trabajar con datos ordenados.

En este capítulo veremos una forma simple de ordenar datos, sin embargo, debemos mencionar que la ordenación de datos es una tarea que se ha estudiado por muchos años y se sigue estudiando, buscando una eficiencia muy alta en el proceso, por lo tanto, existen innumerables métodos de ordenación de datos. El estudio de estos métodos y el análisis de su eficiencia son propios de un texto de Algoritmos y estructuras de datos, por lo que no profundizaremos mucho en este sentido.

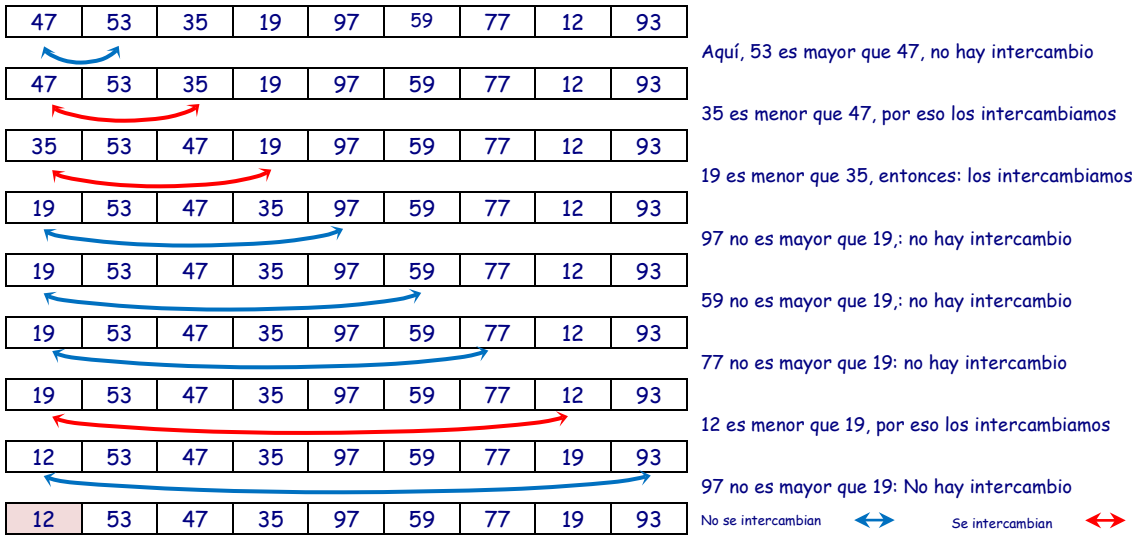
El método presentado a continuación se denomina "Método de intercambio", el nombre se debe a la manera en que se ordenan los datos. Lo primero que haremos será explicar la forma cómo ordenaremos los datos para luego presentar el programa.

Método de intercambio:

Supongamos que tenemos la siguiente secuencia de datos que queremos ordenar:

47	53	35	19	97	59	77	12	93

Este método irá tomando uno a uno los elementos el conjunto comparándolos con el primero, si vemos que el valor que analizamos es menor que el primero, los intercambiamos. De este modo al finalizar el recorrido habremos colocado el menor valor del conjunto en el primer elemento del arreglo. Veamos este proceso:



Al final de este proceso vemos que el menor valor, el 12, terminó colocado en el primer lugar.

Luego tomaremos nuevamente los datos, sin incluir el primero, y procederemos a realizar la misma tarea a partir del segundo elemento. Así colocaremos el segundo valor más pequeño en la segunda casilla:

12	19	53	47	97	59	77	35	93

El proceso continuará hasta haber ordenado todos los datos.

12	19	35	47	53	59	77	93	97

Implementación:

El código del programa debe entonces definir un arreglo en donde se colocarán los datos que se quieren ordenar.

Luego el proceso de ordenación debe recorrer varias veces los elementos del arreglo y en cada recorrido deberá seleccionar el menor valor, dentro de los elementos que aún no se han seleccionado. Estos elementos se colocarán en el arreglo, en orden, desde el inicio. Tenemos que darnos cuenta que en este proceso, luego de analizar al penúltimo elemento, el arreglo ya estará ordenado.

El pseudocódigo para esta tarea sería como se muestra a continuación:

```
Siendo n el número de elementos colocados en el arreglo arr:
Para i = 0 hasta el penúltimo elemento (n-2) hacer
        Colocar en el elemento i del arreglo (arr[i]) el menor valor entre i y n-1
```

La tarea de colocar el menor valor en la casilla analizada (arr[i]) es otro proceso iterativo que va desde el elemento que sigue al que se está trabajando (arr[i+1]) hasta

el último del arreglo. En cada ciclo se deberá tomar un elemento y compararlo con el elemento que se encuentra en la casilla que estamos trabajando (arr[i]); si el elemento que allí se encuentra es mayor que el elemento que hemos tomado, estos valores se deben intercambiar de casillas. Así se conseguirá ordenar el arreglo de menor a mayor valor. El seudocódigo será el siguiente:

```
Para k = i+1 hasta el último elemento (n-1) hacer
      Sí el valor de arr[i] es mayor que el de arr[k] entonces intercambiarlos
```

Según esto, el programa que permita ordenar un conjunto de datos numéricos de menor a mayor será el siguiente:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"
#define MAX_DATOS 100

int main(int argc, char** argv) {
    int arr[MAX_DATOS], numDat;

    leerDatos(arr, numDat, MAX_DATOS);
    cout <<left<<setw(33)<<"Datos en el orden original: ";
    imprimirDatos(arr, numDat);
    ordenaPorIntercambio(arr, numDat);
    cout <<left<<setw(33)<<"Datos ordenados de menor a mayor:";
    imprimirDatos(arr, numDat);

    return 0;
}
```

```
#ifndef FUNCIONESDEARREGLOS_H
#define FUNCIONESDEARREGLOS_H

   void leerDatos(int *, int &, int);
   void imprimirDatos(int *, int );
   void ordenaPorIntercambio(int*, int );
   void cambiar(int &, int &);

#endif /* FUNCIONESDEARREGLOS_H */
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"

void leerDatos(int *arr, int &numDat, const int MAX_DATOS){
    ifstream arch("datos.txt", ios::in);
    int valor;

    if(not arch.is_open()){
        cout << "No se pudo abrir el archivo datos.txt"<<endl;
        exit(1);
    }
    numDat = 0;
    while(1) {
        arch >> valor;
        if(arch.eof())break;

        if(numDat == MAX_DATOS){
            cout<<"ERROR: El arreglo se ha colmado"<<endl;
            cout<<"   se sobepaso el maximo de datos ("<<MAX_DATOS<<")"<<endl;
            exit(1);
        }
        arr[numDat] = valor;
        numDat++;
    }
}
```

```
void imprimirDatos(int *arr, int numDat){
    for(int i=0; i<numDat; i++)
        cout << right << setw(5) << arr[i];
    cout << endl;
}

void ordenaPorIntercambio(int*arr, int numDat){
    for(int i=0; i<numDat-1; i++)
        for(int k=i+1; k<numDat; k++)
            if(arr[i]>arr[k])
                cambiar(arr[i],arr[k]);
}

void cambiar(int &arrI, int &arrK){
    int aux;
    aux = arrI;
    arrI = arrK;
    arrK = aux;
}
```

Al ejecutar el programa con un archivo de texto como se muestra a continuación:

datos.txt

```
35 18 44 21 69 102 8 19 35 33 51 56
```

Se obtendrá el siguiente resultado:

```
Datos en el orden original      :   35  18  44  21  69 102   8  19  35  33  51   56
Datos ordenados de menor a mayor:    8  18  19  21  33  35  35  44  51  56  69  102
```

Adaptación del método de ordenación a problemas más complejos

En este punto veremos cómo, empleando el mismo algoritmo de intercambio, podremos ordenar información que se presente de una manera más compleja.

Por ejemplo, tenemos en un archivo la lista de empleados de una compañía, en la que se aprecie el código, teléfono y sueldo de cada empleado y queremos ordenarla por el código del empleado. El archivo puede ser similar al siguiente:

empleados.txt

```
78392844    990755432     2865.25
63888343    977001901    13843.66
93619155    999553294      416.56
26701964    993417886     4543.48
…           …             …
```

Aquí, el programa deberá definir tres arreglos y en cada uno debe colocar los datos de los empleados, vale decir, definir un arreglo de enteros para colocar los códigos, un arreglo de enteros para los teléfonos y otro de punto flotante para los sueldos.

El problema aquí es que, si en el proceso de ordenación involucramos solo un arreglo, el arreglo de códigos, obtendremos una repuesta equivocada. La razón para esto es que la ordenación hará que los datos en el arreglo de códigos se alteren, cambiando los datos de posición, sin embargo, los otros datos, teléfono y sueldo, permanecerán en su lugar inicial, lo que producirá que los datos ya no concuerden entre sí, por lo que el reporte que se generé será incoherente.

Según esto, el programa que permita ordenar estos registros del archivo será el siguiente:

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"
#define MAX_DATOS  100
```

```
int main(int argc, char** argv) {
    int codigo[MAX_DATOS], telefono[MAX_DATOS], numDat;
    double sueldo[MAX_DATOS];
   // Se colocarán dos reportes en un archivo
    ofstream archRep("ReporteDePersonal.txt", ios::out);
    if (not archRep.is_open()){
        cout<<left<<"ERROR: No se pudo abrir el archivo Reporte.txt"<<endl;
        exit(1);
    }
    leerDatos(codigo, telefono, sueldo, numDat);
    archRep<<left<<"Datos en el orden original:"<<endl;
    imprimirDatos(archRep, codigo, telefono, sueldo, numDat);
    ordenarPorIntercambio(codigo, telefono, sueldo, numDat);
    archRep<<endl<<left<<"Datos ordenados de menor a mayor:"<<endl;
    imprimirDatos(archRep, codigo, telefono, sueldo, numDat);
    return 0;
}
```

```
#ifndef FUNCIONESDEARREGLOS_H
#define FUNCIONESDEARREGLOS_H
   void leerDatos(int *, int *,double*, int &);
   void imprimirDatos(ofstream &, int *,int*,double*, int );
   void ordenarPorIntercambio(int*,int*,double*, int );
   void cambiarInt(int &, int &);
   void cambiarDbl(double & , double &);
#endif /* FUNCIONESDEARREGLOS_H */
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"
void leerDatos(int *codigo, int *telefono, double *sueldo, int &numDat){
    ifstream arch("PersonasConVariosCampos.txt", ios::in);
    if(not arch.is_open()){
       cout<<left<<"No se pudo abrir el archivo PersonasConVariosCampos.txt"<<endl;
        exit(1);
    }
    int cod, tel;
    double sue;

    numDat = 0;
    while(1) {
        arch >> cod >> tel >> sue;
        if(arch.eof())break;
        codigo[numDat] = cod;
        telefono[numDat] = tel;
        sueldo[numDat] = sue;
        numDat++;
    }
}
void imprimirDatos(ofstream &archRep, int *codigo, int *telefono, double *sueldo,
                   int numDat){
    archRep.precision(2);
    archRep<<fixed;
    archRep <<right<<setw(9)<<"CODIGO"<<setw(12)<<"TELEFONO"
            <<setw(12)<<"SUELDO"<<endl;
    for(int i=0; i<numDat; i++)
        archRep<<right<<setw(10)<<codigo[i]<<setw(12)<<telefono[i]
               <<setw(12)<<sueldo[i]<<endl;
    archRep<<endl;
}
void ordenarPorIntercambio(int *codigo, int *telefono, double *sueldo, int numDat){
    for(int i = 0; i<numDat-1; i++)
        for(int k=i+1; k<numDat; k++)
            if(codigo[i]>codigo[k]){
                cambiarInt(codigo[i], codigo[k]);
                cambiarInt(telefono[i], telefono[k]);
                cambiarDbl(sueldo[i], sueldo[k]);
            }
}
```

```
void cambiarInt(int &arrI, int &arrK){
    int aux;
    aux = arrI;
    arrI = arrK;
    arrK = aux;
}

void cambiarDbl(double &arrI, double &arrK){
    double aux;
    aux = arrI;
    arrI = arrK;
    arrK = aux;
}
```

Con esto se obtiene la siguiente respuesta, observe que la correlación de los datos se mantiene:

Reporte.txt

```
Datos en el orden original:
   CODIGO      TELEFONO       SUELDO
  78392844     990755432      2865.25
  63888343     977001901     13843.66
  93619155     999553294      8416.56
  26701964     993417886      4543.48
  45051489     961094675      4409.70
  …            …              …
  59985330     991880607       631.04
  94037877     980915507     11329.25
  42977551     980332962       864.00
  22530045     964497199     11335.51

Datos ordenados de menor a mayor:
   CODIGO      TELEFONO       SUELDO
  11814159     994824461      5658.42
  11815912     972316133      7996.73
  13557075     975463589      3934.43
  22530045     964497199     11335.51
  25704775     978038682      7737.68
  …            …              …
  94037877     980915507     11329.25
  94355762     987895728      2668.90
  96481146     978274059      5688.46
  98070924     999095733     11702.55
```

Un segundo caso que permite ver una situación compleja en la ordenación se da cuando pueden existir datos repetidos en el campo que se elige como criterio de ordenación. Por ejemplo, si tuviéramos un archivo en el que se encuentren los datos de muchísimas personas, donde se tiene allí el DNI, su sueldo, el código de la región y el código de la ciudad donde vive, como se muestra a continuación:

empleados.txt

```
93619155    8104.62   4   11
29543877   11606.74   1   12
81711999   12982.21   5   12
78868406     223.76   4   14
98070924    2824.29   1   13
26701964    2094.25   3   12
82223817    4636.64   1   12
42977551    8092.32   1   11
84751690   14466.57   1   12
       …          …   …    …
59985330    2826.73   3   15
84629106    4791.57   5   15
63888343    6279.96   2   14
27055708    3799.53   2   15
41212003    4198.62   3   11
94355762    5903.14   1   10
82924309    1177.69   5   14
       …          …   …    …
```

Si ordenáramos estos datos por la región donde vive la persona, intercambiando todos los campos como en el problema anterior, podemos observar, como se muestra a continuación, que la respuesta del programa no es del todo satisfactoria:

Reporte.txt

```
  29543877     1    12   11606.74
  98070924     1    13    2824.29
  82223817     1    12    4636.64
  42977551     1    11    8092.32
  84751690     1    12   14466.57
  41427373     1    12   12665.02
  94355762     1    10    5903.14
  77314106     1    14    8534.76
         …     …     …          …
  56836804     2    12   14834.08
  27055708     2    15    3799.53
  63888343     2    14    6279.96
  40628395     2    14    5516.81
         …     …     …          …
  22530045     3    11    8751.84
  34169256     3    14     582.46
  59985330     3    15    2826.73
  13557075     3    12    2169.85
         …     …     …          …
  33609304     5    13    6563.13
  45051489     5    11    9930.04
  81711999     5    12   12982.21
  27067084     5    14   11528.51
  82924309     5    14    1177.69
  83459747     5    15     709.44
         …     …     …          …
```

Lo primero que se puede observar es que los datos se han agrupado, y ordenado, por la región. Sin embargo, hay algo que no está bien, si nos fijamos en un grupo de personas con la misma región, podemos darnos cuenta que allí no hay orden. Ubicar una persona, conociendo la región donde vive y su DNI, sería muy complicado debido a la gran cantidad de personas que puede vivir en una región. Por esto, dentro de una misma región deberíamos agrupar, en orden, a las que coinciden con el mismo código de la ciudad donde viven, y dentro de la que tiene el mismo código de región y ciudad, por su DNI.

Para lograr esto debemos analizar el algoritmo que empleamos para ordenar. Cuando el algoritmo hace la pregunta: **`if(arr[i]>arr[k])`**, lo que se está haciendo es verificar si el elemento **`arr[i]`** está ordenado en relación con el elemento **`arr[k]`**. Por ejemplo, si estuviéramos ordenando de menor a mayor y en **`arr[i]`** tuviéramos el valor **`1111111`** y en **`arr[k]`**el valor **`5555555`**, no habría que intercambiarlos ya que **`1111111`** esta ordenado con respecto a **`555555`**. Pero si en **`arr[i]`** tuviéramos el valor **`777777`** y en **`arr[k]`** el valor **`5555555`**, si se debería intercambiar porque no están ordenados relativamente.

Entonces cuando los ordenemos alfabéticamente, empleado este criterio, debemos determinar cuándo dos datos deben intercambiarse. Si una persona **`i`** viviera en la región **`4`** y la persona **`k`** viviera en la región **`2`**, sería suficiente con este dato para saber que debemos intercambiarlos.

Sin embargo, si la persona **i** viviera en la región **3** y la persona **k** también viviera en la región **3**, no podemos tomar una decisión para determinar si deben o no intercambiarse. Entonces debemos pensar en una condición más compleja que permita solucionar esta situación, la pregunta debería llevar a verificar que si las regiones son iguales se debe hacer el intercambio si el código de la ciudad donde viven no está en orden. A esto se debe agregar que en el caso que ambos datos, región y ciudad, sean iguales, se deberán intercambiar si no están ordenados por el DNI. Estos tres casos se deben expresar en una sola condición.

El programa que resolverá este problema es el siguiente:

```
#include "FuncionesDeArreglos.h"
#define MAX_DATOS  100

int main(int argc, char** argv) {
    int codigo[MAX_DATOS], region[MAX_DATOS],ciudad[MAX_DATOS],numDat;
    double sueldo[MAX_DATOS];

    leerDatos(codigo, region, ciudad, sueldo, numDat);
    ordenarPorIntercambio(codigo, region, ciudad, sueldo, numDat);
    imprimirDatos(codigo, region, ciudad, sueldo, numDat);

    return 0;
}
```

```
#ifndef FUNCIONESDEARREGLOS_H
#define FUNCIONESDEARREGLOS_H

void leerDatos(int *, int *, int *, double*, int &);
void imprimirDatos(int *,int*, int *, double*, int);
void ordenarPorIntercambio(int*,int*, int *, double*, int );
void cambiarInt(int &, int &);
void cambiarDbl(double & , double &);

#endif /* FUNCIONESDEARREGLOS_H */
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"

void leerDatos(int *codigo, int *region, int *ciudad, double *sueldo, int &numDat){
    ifstream arch("PersonasDatosRepetidos.txt", ios::in);
    if(not arch.is_open()){
        cout<<left<<"No se pudo abrir el archivo PersonasDatosRepetidos.txt"<<endl;
        exit(1);
    }
    int cod, reg, ciu;
    double sue;

    numDat = 0;
    while(1) {
        arch >> cod >> sue >> reg >> ciu;
        if(arch.eof())break;
        codigo[numDat] = cod;
        region[numDat] = reg;
        ciudad[numDat] = ciu;
        sueldo[numDat] = sue;
        numDat++;
    }
}

void imprimirDatos(int *codigo,int *region,int *ciudad,double *sueldo,int numDat){
    ofstream archRep("ReporteDePersonal.txt", ios::out);
    if (not archRep.is_open()){
        cout<<left<<"ERROR: No se pudo abrir el archivo Reporte.txt"<<endl;
        exit(1);
    }
```

```
        archRep.precision(2);
        archRep<<fixed;
        archRep<<left<<"Datos ordenados de menor a mayor:"<<endl;
        archRep <<right<<setw(9)<<"CODIGO"<<setw(9)<<"REGION"<<setw(9)<<"CIUDAD"
                <<setw(9)<<"SUELDO"<<endl;
        for(int i=0; i<numDat; i++)
            archRep<<right<<setw(10)<<codigo[i]<<setw(6)<<region[i]
                   <<setw(9)<<ciudad[i]<<setw(12)<<sueldo[i]<<endl;
        archRep<<endl;
    }

    void ordenarPorIntercambio(int *codigo, int *region, int *ciudad,
                               double *sueldo, int numDat){
        for(int i = 0; i<numDat-1; i++)
          for(int k=i+1; k<numDat; k++)
            if(region[i]>region[k]
                  or
               region[i] == region[k] and ciudad[i]>ciudad[k]
                  or
               region[i] == region[k] and ciudad[i]==ciudad[k] and codigo[i]>codigo[k]){
                 cambiarInt(codigo[i], codigo[k]);
                 cambiarInt(region[i], region[k]);
                 cambiarInt(ciudad[i], ciudad[k]);
                 cambiarDbl(sueldo[i], sueldo[k]);
            }
    }

    void cambiarInt(int &arrI, int &arrK){
        int aux;
        aux = arrI;
        arrI = arrK;
        arrK = aux;
    }

    void cambiarDbl(double &arrI, double &arrK){
        double aux;
        aux = arrI;
        arrI = arrK;
        arrK = aux;
    }
```

El resultado de este programa será el siguiente:

Reporte.txt

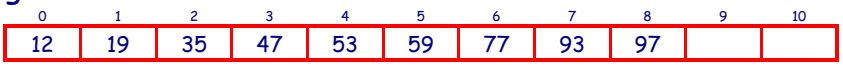
CODIGO	REGION	CIUDAD	SUELDO
…	…	…	…
29543877	1	12	11606.74
41427373	1	12	12665.02
54412597	1	12	2032.61
82223817	1	12	4636.64
84751690	1	12	14466.57
33521881	1	13	5678.90
…	…	…	…
22530045	3	11	8751.84
41212003	3	11	4198.62
13557075	3	12	2169.85
26701964	3	12	2094.25
…	…	…	…
45717042	4	11	9902.02
58268885	4	11	2414.42
93619155	4	11	8104.62
94037877	4	11	9217.16
78868406	4	14	223.76
…	…	…	…
45051489	5	11	9930.04
78392844	5	11	5408.55
81711999	5	11	12982.21
11815912	5	13	11452.44
…	…	…	…

Como se ve, es lo que esperábamos.

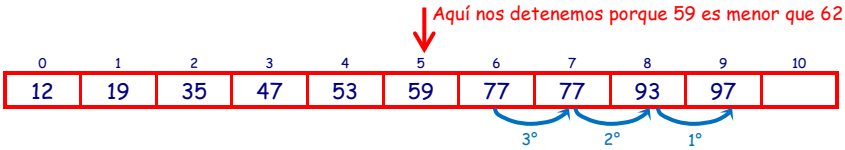
Método de inserción:

Este método se emplea cuando tenemos un arreglo ordenado y queremos agregarle un nuevo dato. Si empleáramos el método de intercambio tendríamos primero que colocar el dato al final del arreglo y luego ordenarlo aplicando el método, sin embargo, si analizamos un poco el método de intercambio veremos que el tiempo que demoraría en ordenar estos datos bajo estas condiciones sería el mismo que si todos los datos estuvieran desordenados. Si cada cierto tiempo hubiera que agregar un nuevo dato al arreglo, esta tarea sería muy ineficiente, por eso vamos a mostrar a continuación otro método denominado "de Inserción" el cual, para estas situaciones resulta ser mucho más eficiente que el método de intercambio y que algunos otros métodos más. El método lo describiremos a continuación.

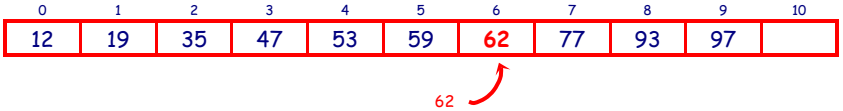
Supongamos que queremos agregar el valor 62 en un arreglo que ya tenemos ordenado con los datos siguientes:



El método consistirá entonces en buscar, desplazando los datos desde el último, hasta encontrar la ubicación del valor de manera que el arreglo quede ordenado al final. Esto es:



Finalmente se coloca el nuevo dato en la posición correcta. Esto es:



A continuación, presentamos un programa que ordenará empleando este método los datos leídos desde un archivo de textos.

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"
#define MAX_DATOS 100

int main(int argc, char** argv) {
    int arr[MAX_DATOS], numDat;

    // La lectura y ordenación se hacen en el mismo proceso porque cada vez
    // que se lee un dato se inserta en el arreglo de manera que quede ordenado.
    leerYOrdenarDatosPorInsecion(arr, numDat);
    imprimirDatos(arr,numDat);

    return 0;
}
```

```
#ifndef FUNCIONESDEARREGLOS_H
#define FUNCIONESDEARREGLOS_H

void leerYOrdenarDatosPorInsecion(int *, int &);
void imprimirDatos(int*, int);
void insertar(int, int*, int&);

#endif /* FUNCIONESDEARREGLOS_H */
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "FuncionesDeArreglos.h"
```

```
void leerYOrdenarDatosPorInsecion(int *arr, int &numDat){
    ifstream arch("datos.txt", ios::in);
    int dato;
    if(not arch.is_open()){
        cout<<left<<"No se pudo abrir el archivo datos.txt"<<endl;
        exit(1);
    }
    numDat = 0;
    while (1){
        arch>>dato;
        if(arch.eof())break;
        insertar(dato, arr, numDat);
    }
}

void insertar(int dato, int*arr, int &numDat){
    int i;
    // Primero nos colocamos al final de los datos.
    i = numDat;
    // Luego buscamos la posición del nuevo dato, desplazando los otros
    while (1){
        i--;
        if (i<0 or arr[i]<dato) break;
        arr[i+1] = arr[i];  // Desplazamos los valores del arreglo una posición
    }
    arr[i+1] = dato;  // Colocamos el dato en la posición correcta
    numDat++;
}

void imprimirDatos(int *arr, int numDat){
    ofstream archRep("ReporteDeValores.txt", ios::out);
    if (not archRep.is_open()){
        cout<<left<<"ERROR: No se pudo abrir el archivo Reporte.txt"<<endl;
        exit(1);
    }
    for(int i=0; i<numDat; i++)
        archRep<<right<<setw(10)<<arr[i];
    archRep<<endl;
}
```

Este método se puede emplear de manera similar a lo que hicimos en el método de intercambio para ordenar datos más complejos. Se sugiere que usted desarrolle estas modificaciones.

Búsqueda de datos en un arreglo

Las operaciones de búsqueda en arreglos es una tarea que se realiza y se repite muchas veces en la mayoría de aplicaciones. En los capítulos anteriores hemos desarrollado aplicaciones que buscan datos en un archivo de texto, este capítulo vamos a desarrollar esta operación con más profundidad y ahora aplicándola en arreglos. Además, revisaremos los métodos de búsqueda secuencial y búsqueda binaria, comparando su eficiencia.

Búsqueda secuencial:

El método secuencial de búsqueda es un proceso muy simple, cuando en un archivo de textos leemos una a una las líneas buscando un dato, estamos realizando un proceso de búsqueda secuencial.

Cuando trabajamos con arreglos también se va a requerir hacer operaciones de búsqueda. Normalmente esta tarea se realiza cuando se tienen varios arreglos que contienen diferente información, por ejemplo: DNI, teléfono, sueldo, etc. de diferentes personas, si luego se quiere conocer los datos de una persona teniendo por ejemplo su DNI.

La operación de búsqueda se realizará en este caso mediante una función que recibirá el dato a buscar, en este caso el DNI de una persona, y el arreglo que contiene todos DNI de las personas (no se requieren los demás arreglos), la función devolverá la posición del DNI buscado en el arreglo de modo que con esa posición podamos conocer los restantes datos. Es muy importante que la función contemple la posibilidad de no encontrar el dato en el arreglo por lo que la función deberá un valor que lo indique.

El código de una función típica de búsqueda que aplique el método secuencial se muestra a continuación:

```
int buscarSecuencialmente(int dato, int *arr, int numDat){
   for(int pos=0; pos<numDat; pos++)
      if(d == dni[pos]) return pos; //Devolvemos la posición
   return -1; // No se encontró
}
```

Como se observa en el código, la función devuelve la posición donde se encuentra el dato buscado, pero también, como hemos indicado, devuelve el valor -1 que indicará que el dato no se encuentra en el arreglo, la función nuca debe suponer que siempre se encontrará el dato buscado.

Otra cosa que podemos ver es que el código empleado es muy simple, sin embargo, resulta ser un método muy ineficiente. La razón para esta suposición es que este proceso equivale a buscar un teléfono en una guía telefónica y proceder a leer los datos de los usuarios uno a uno desde la primera página. Entonces, ¿por qué dedicamos tiempo a estudiarlo e implementarlo?, la respuesta es igual de simple, si los datos estuvieran desordenados y no pudiéramos ordenarlos, esta sería la única forma de buscar un elemento allí.

Midamos ahora la eficiencia de este método para poder compararlo luego con otros. Para hacerlo, emplearemos una forma sencilla, vamos a determinar, en promedio, cuántos ciclos tendríamos que realizar para poder encontrar un dato en el arreglo. Esto lo vamos a hacer determinando dos casos:

- Caso más favorable: La situación más favorable en la búsqueda es en la que el dato que estamos buscando se encuentre en el primer elemento del arreglo, por lo tanto, sólo tendremos que hacer un ciclo en la búsqueda. Entonces anotaremos que en el caso más favorable será cuando $C_{más\_fav} = 1$.
- Caso menos favorable: De manera similar la situación menos favorable se dará cuando el dato buscado se encuentre en el último elemento del arreglo, entonces tendremos que hacer tantos ciclos como elementos (n) tenga el arreglo. Por lo tanto, el caso menos favorable será cuando: $C_{menos\_fav} = n$.

De acuerdo a estas conjeturas, podemos afirmar que en promedio la cantidad de ciclos que tendremos que hacer buscando secuencialmente en un arreglo estará dado por:

$$C_{promedio} = \frac{(1+n)}{2}$$

siendo n el número de elementos del arreglo

Búsqueda binaria:

Cuando se quiere encontrar el número telefónico de una persona en la guía telefónica, a nadie se le ocurriría comenzar a buscarlo desde la primera página, leyendo uno por uno los nombres que allí estén, en otras palabras, es ilógico emplear un método secuencial para realizar esta tarea. Sin embargo, si la guía telefónica no estuviera ordenada por el nombre del usuario o si lo que buscamos es el nombre del usuario teniendo su número telefónico, no nos quedaría otra opción que emplear el método secuencial. Con esto estoy diciendo que no se puede despreciar un método, siempre que un método se emplee en condiciones adecuadas será un buen método. Por ejemplo, se queremos buscar un dato en un archivo de textos, no se podrá emplear otro método que no sea el secuencial, esto debido a la naturaleza de los archivos de textos.

La búsqueda binaria es un método muy eficiente para realizar búsquedas, sin embargo el método no sirve si es que los datos no están ordenados de acuerdo a la información que se está buscando, esto es, si tenemos el DNI de una persona y queremos conocer su nombre, los datos deben estar ordenados por el DNI para poder emplear este método, si no está ordenado o está ordenado por otro campo, por ejemplo el nombre, el método no se podría emplear.

El método de búsqueda binaria, al igual que muchos otros métodos, se basa en la forma cómo se realizan ciertos procesos manualmente. A continuación, analizaremos este proceso.

Pensemos en la forma cómo se busca un teléfono en la guía telefónica, si buscamos el teléfono de la señora "Ana Cecilia Roncal Neyra" cogemos la guía y la abrimos por la mitad aproximadamente, luego observamos los nombres que aparecen en esas páginas, si notamos que los nombres corresponde a la letra "L" querrá decir que el nombre que buscamos no estará en la parte izquierda de la guía si no que estará a su derecha, por eso concentramos nuestra búsqueda en la parte derecha descartando por completo la izquierda. Luego se toma la parte derecha y se vuelve a abrir tratando de acercarnos al nombre. Si encontramos la letra "T" sabremos que está en la parte izquierda. El proceso se repite hasta encontrar a la persona buscada. Lo que hemos hecho no es otra cosa que emplear un método binario para buscar, binario porque separamos en cada ciclo los datos en dos.

Formalicemos el método empleando un arreglo como el que se muestra a continuación:

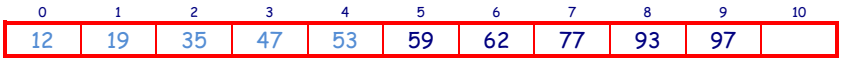
0	1	2	3	4	5	6	7	8	9	10
12	19	35	47	53	59	62	77	93	97	

Como se puede apreciar el arreglo está ordenado, de lo contrario no podríamos emplear el método. Ahora si buscáramos la posición donde se encuentra el número 62, lo que tenemos que hacer es dividir los datos en dos, para esto, como el proceso debe ser automatizado, no podemos definir un punto de separación basándonos en una posición cercana al dato como lo hacemos manualmente, por eso determinaremos siempre un punto medio.

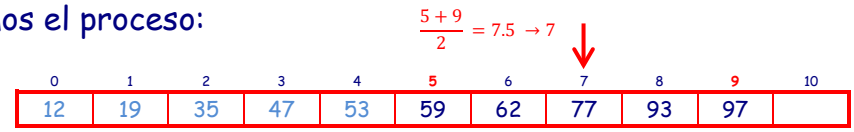
$\frac{0+9}{2} = 4.5 \rightarrow 4$ Aquí nos detenemos porque 59 es menor que 62

0	1	2	3	4	5	6	7	8	9	10
12	19	35	47	53	59	62	77	93	97	

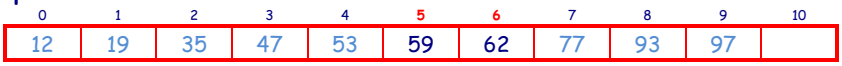
Luego observemos qué valor se encuentra en la posición calculada, en este caso el número 53. Como nos damos cuenta que este valor no es el que buscamos y es menor que el valor buscado (62) sabremos que el dato está a la derecha, por lo que descartamos de manera lógica (no física), la parte izquierda.



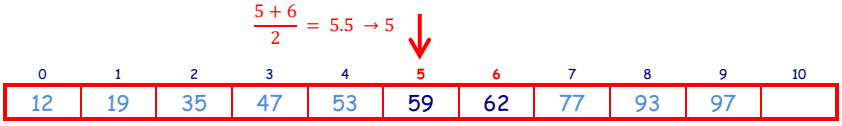
Luego repetimos el proceso:



El valor 77 no es el que buscamos y es mayor que 62, por lo que descartamos la parte derecha del grupo definido.

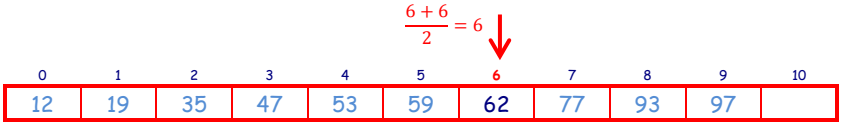


Y seguimos:



Luego:

Finalmente:



Hallamos el valor.

Observe que hemos empleado cuatro ciclos en lugar de los siete que hubiéramos empleado con el método binario.

Por otro lado, si el dato buscado hubiera sido 63 y no 62, ¿cómo llegamos a la conclusión que el dato no se encuentra en el arreglo?, la solución es simple, inicialmente se definen los límites en donde se realizará la búsqueda en el arreglo, el límite inferior será 0 y el superior el índice del último elemento guardado en el arreglo (n-1). Si luego del primer ciclo, cuando determinamos el punto medio (pm) vemos que, si el dato en esa posición no es el que buscamos, entonces debemos modificar los límites de búsqueda, así, si se determina que el dato se encuentra a la izquierda, los nuevos límites serán 0 y pm-1, y si está a la derecha los límites serán pm+1 y n-1. Si se sigue procesando el algoritmo, por el hecho que al punto medio se le suma o resta una unidad se va a llegar a un punto en el que el límite inferior se vuelve mayor que el superior, es en ese instante cuando de determina que el dato no se encuentra en el arreglo.

El código de una función típica de búsqueda que aplique el método binario se muestra a continuación:

```
int buscarDeManeraBinaria(int dato, int *arr, int numDat){
    int limiteInf = 0, limiteSup = numDat - 1, puntoMedio;

    while(1){
        if(limiteInf>limiteSup) return -1;     //No lo encontró
        puntoMedio = (limiteInf+limiteSup)/2;
```

```
            if(dato == arr[puntoMedio])
                return puntoMedio;          //Devolvemos la posición
            if(dato > arr[puntoMedio])
                limiteInf = puntoMedio + 1;
            else
                limiteSup = puntoMedio - 1;
        }
    }
```

Ahora vamos a determinar, como lo hicimos en el otro método, la cantidad de ciclos que tendremos que hacer, en promedio, para encontrar un dato con este método.

- Caso más favorable: La situación más favorable es en la que el dato que estamos buscando esté en medio del arreglo, por lo tanto, sólo tendremos que hacer un ciclo en la búsqueda. Entonces anotaremos que en el caso más favorable será cuando $C_{más\_fav} = 1$.
- Caso menos favorable: Se puede calcular de la siguiente manera:
  - Si luego del 1er. ciclo no se encuentra el elemento, entonces el número de elementos se reduce a $\frac{n}{2}$.
  - Luego del 2do., si tampoco lo encontramos, nos quedamos ahora con $\frac{n}{2\times2}$ elementos.
  - Si seguimos así los elementos se reducirán hasta llegar a ser $\frac{n}{2\times2\times2\times\cdots\times2} = 1$
    Si "p" es el número de ciclos, entonces tenemos que $\frac{n}{2^p} = 1$
    Despejando "p" tendremos:

$$\rightarrow n = 2^p$$
$$\rightarrow log_2(n) = log_2(2^p)$$
$$\rightarrow log_2(n) = p \times log_2(2)$$
$$\rightarrow log_2(n) = p \times 1$$
$$\rightarrow log_2(n) = p$$

    Por lo tanto, podemos afirmar que $C_{menos\_fav} = \log_2(n)$.

Entonces, podemos afirmar que en promedio la cantidad de ciclos que tendremos que hacer buscando de manera binaria en un arreglo estará dado por:

$$C_{promedio} = \frac{(1 + log_2(n))}{2}$$

siendo n el número de elementos del arreglo

Ahora ya podemos comparar el método de búsqueda secuencial con el método de búsqueda binaria. La tabla siguiente muestra esta comparación:

	Búsqueda secuencial	Búsqueda binaria
n	$C_{promedio} = \frac{(1+n)}{2}$	$C_{promedio} = \frac{(1+log_2(n))}{2}$
10	5.5	2.2
100	50.5	3.8
1,000	500.5	5.5
10,000	5,000.5	7.1
100,000	50,000.5	8.8.
1'000,000	500,000.5	10.5
10'000,000	5'000,000.5	12.1

Observe que mientras en la búsqueda secuencial se requieren en promedio 5 millones de ciclos cuando hay 10 millones de datos, sólo se requieren un poco más de 12 ciclos en la búsqueda binaria. Es claro cuál de los métodos es el más eficiente.

Cómo desordenar datos

A pesar que esto parezca trivial, desordenar datos puede ser una tarea muy útil y compleja. Por ejemplo, cuando se requiera una secuencia aleatoria de valores que no se repitan, esta situación se puede encontrar por ejemplo en la simulación de un juego de cartas, en donde la desordenación se refiere a barajar las cartas.

Si piensa que esta tarea es fácil, y que sólo tiene que ejecutar una función aleatoria varias veces, verá que esto no es así.

Primero veamos cómo se determina un valor aleatorio en el lenguaje C++. En la biblioteca cstdlib se definen dos funciones y una constante simbólica para este fin:

- `rand()`
- `srand(unsigned int)`
- `RAND_MAX`

La función rand() devuelve un valor entero pseudoaleatorio entre 0 y RAND_MAX, donde RAND_MAX guarda el valor 32767. Se dice que el valor es pseudoaleatorio porque no se puede decir que es puramente aleatorio ya que el resultado es producto de una expresión que sigue siempre los mismos pasos.

Entonces, si queremos obtener una lista de 10 valores "aleatorios" entre 1 y 20, deberíamos escribir el siguiente código:

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

int main(int argc, char** argv) {
    int r;
    for(int i=0; i < 10; i++){
        r = rand()%20 + 1;
        cout << setw(4) << r;
    }
    cout<<endl;
    cout << RAND_MAX << endl;
    return 0;
}
```

El programa dará como resultado lo siguiente:

```
   2   8  15   1  10   5  19  19   3   5
```

Podemos observar en este resultado algunas cosas interesantes, la primera es que los valores se pueden repetir, de esta forma, tendremos que estudiar otra forma para garantizar que todos los valores sean diferentes. La segunda y más importante es que si el programa se ejecuta varias veces, se obtendrán exactamente los mismos resultados, esto pasa porque los valores generados no son realmente aleatorios, por eso la referencia a pseudoaleatorios.

Como dijimos, los valores se generan empleando una expresión en la que su evaluación parte de un mismo dato inicial denominado "semilla" (seed), si el valor de ese dato cambiase, la secuencia de valores resultantes sería diferente. Este valor se puede

modificar mediante la función srand. Esta función recibe como parámetro un valor entero que servirá para cambiar el valor de la semilla.

El código siguiente muestra la forma cómo emplearla:

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

int main(int argc, char** argv) {
    int r;
    srand(5);
    for(int i=0; i < 10; i++){
        r = rand()%20 + 1;
        cout << setw(4) << r;
    }
    cout<<endl;
    cout << RAND_MAX << endl;
    return 0;
}
```

Obtendrá el siguiente resultado:

```
  15  14  16  10   1  11   8  10  13   1
```

Sin embargo, la función srand no soluciona completamente el problema, ya que siempre dará la misma secuencia para el valor de la semilla 5. Lo que se debe hacer es procurar que el parámetro que se introduzca a la función srand cambie cada vez que se ejecute el código, tarea complicada. Afortunadamente el lenguaje C posee una gran cantidad de bibliotecas de funciones y entre ellas se encuentra la biblioteca time.h, en la que se define la función time (), al usarse como: time (nullptr), la función devolverá el número de segundos que han trascurrido desde las 00:00 horas del 1 de enero de 1970 hasta el momento en que se ejecuta el código[1]. Luego, el resultado de esta función se puede pasar como parámetro de la función srand (), con lo que se podrá garantizar que cada vez que el código se ejecute, la semilla será diferente.

Finalmente, el código para generar los 10 valores aleatorios será el que se muestra a continuación:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <cstdlib>
#include <ctime>

int main(int argc, char** argv) {
    int r;
    srand(time(nullptr));
    for(int i=0; i < 10; i++){
        r = rand()%20 + 1;
        cout << setw(4) << r;
    }
    cout<<endl;
    cout << RAND_MAX << endl;
    return 0;
}
```

Solucionado este problema, ahora tenemos que concentrarnos en conseguir que la secuencia que se obtenga no aparezca valores repetidos.

[1] Información obtenida de: http://www.cplusplus.com/reference/ctime/time/

Se podría pensar que para lograr una lista de datos desordenada en la que no se repitan los valores se tendría que generar el valor aleatorio y luego verificar si no ha salido, si ya salió se deberá generar un nuevo valor. Sin embargo, como veremos a continuación esta no es una buena solución debido a que conforme vamos obteniendo nuevos valores, la probabilidad que es siguiente se repita cada vez será más alta. Por lo que para conseguir los últimos valores se deberán generar muchísimos intentos.

El siguiente programa muestra este algoritmo, y para ver la eficiencia hemos colocado un contador que nos dirá cuántas veces se ejecutará una instrucción crítica en el programa.

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <cstdlib>
#include <ctime>
#include "FuncionesDeArreglos.h"
#define MAX_DATOS 100

int main(int argc, char** argv) {
    int nCiclos = 0, arr[MAX_DATOS] {};

    srand(time(nullptr));
    for (int pos=0; pos<MAX_DATOS; pos++)
        generaValor(arr, pos, nCiclos);

    for (int i=0; i<MAX_DATOS; i++)
        cout<<right<<setw(4)<<arr[i];
    cout<<endl;
    cout<<left<<"Numero de ciclos: "<<nCiclos<<endl;
    return 0;
}
```

```
#ifndef FUNCIONESDEARREGLOS_H
#define FUNCIONESDEARREGLOS_H

  void generaValor(int*, int, int&);
  bool noSeRepite(int, int*, int, int&);

#endif /* FUNCIONESDEARREGLOS_H */
```

```
#include <cstdlib>
#include "FuncionesDeArreglos.h"
#define MAX_DATOS 100

void generaValor(int *arr, int pos, int &nCiclos){
    int valor;
    while (1){
        valor = rand()% MAX_DATOS + 1;
        if (noSeRepite(valor, arr, pos, nCiclos)) break;
    }
    arr[pos] = valor;
}

bool noSeRepite(int valor, int*arr, int pos, int&nCiclos){
    int k = 0;
    bool esta = false; // false indica que no está
    while(1){
        if(k == pos or esta) break;
        nCiclos++;
        esta = arr[k] == valor;
        k++;
    }
    return  not esta;
}
```

Al ejecutar será similar al siguiente:

```
  48   9  36  50  75  26  65  34  41  25  63 100  14  54  51  79  68  53  16  56  55
  86  39  15   8   3  87  38   7  99  64  88  44  10  97  22  58  45  74  76  20  80
  47  40  21  46  35  95  82  11  91  43   5   4   1  71  17  70  83  27  28  78  69
  57  90  52  96  23  84   2  24  94  32  85  12  33  73  61  60  77  29  19  13  49
  98  93  62  66   6  92  59  67  89  31  81  37  72  42  30  18
Número de ciclos: 42322
```

Como se puede observar el algoritmo obtiene el resultado esperado sin embargo el costo ha sido muy alto, 42,322 veces se ejecutó la instrucción más crítica. Este valor puede ser más grande aún, de acuerdo a la secuencia de valores que vayan saliendo. Esto no se puede aceptar.

Ahora veremos un algoritmo muy ingenioso, pero extremadamente eficiente que aplica el concepto de desordenar para generar una lista de valores sin repetir.

El algoritmo empieza llenando un arreglo con valores consecutivos, y a partir de allí los desordena. Luego de esto, genera un valor aleatorio entre 0 y el tamaño del arreglo menos 1. El valor aleatorio se toma como una posición en el arreglo. Seguidamente se intercambia el último valor del arreglo con el de la posición obtenida con el valor aleatorio. Finalmente se repite el mismo proceso, pero ahora se toman todos los datos del arreglo sin considerar el último. Debido al intercambio de valores, no importará que el valor aleatorio se repita ya que en esa posición no se encontrará el mismo valor. Este proceso se repite hasta que sólo quede un elemento por analizar. De esta manera se recorre una sola vez el arreglo y se conseguirá que la secuencia esté desordenada aleatoriamente.

A continuación, se presenta el programa:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <stdlib.h>
#include <time.h>
#define MAX_DATOS 100

int main(int argc, char** argv) {
    int arr[MAX_DATOS], pos, aux, nCiclos = 0;
    srand(time(nullptr));

    // Llenamos el arreglo con valores consecutivos
    for(int i= 0; i<MAX_DATOS; i++) arr[i] = i+1;
    // Empezamos a desordenar
    for (int limite = MAX_DATOS-1; limite>0; limite--) {
        // Determinamos un valor aleatorio entre uno y el límite
        pos = rand()% MAX_DATOS ;
        aux = arr[limite]; //Intercambiamos los datos de esas posiciones
        arr[limite] = arr[pos];
        arr[pos] = aux;
        nCiclos++; //Instrucciones críticas
    }
    //Imprimimos los valores
    for(int i= 0; i<MAX_DATOS; i++)
        cout<<right<<setw(4)<<arr[i];
    cout<<endl;
    cout<<left<<"Numero de ciclos: "<<nCiclos<<endl;

    return 0;
}
```

El resultado es el siguiente:

```
  95  75  47  11  97  66  16  40  59  29  54  94  65  36   5  15  90  31  22  76   2
  55  43  21  18  12  42  74  99  34  81  80  24  10  27  46  25  37  84  83  78  38
  58  30  98  41  14  85  13  86  61  92  56  63   4  19  87  49  67   3  50  57  52
  69  44  32  68  48  45  26  51  17  62  20   1  82  53 100  23   6  71   9  73  33
  64  60  96  39  77  70  88   7  79  28   8  91  89  35  93  72
Número de ciclos: 99
```

Observe que la cantidad de ciclos equivale a la cantidad de datos que se quiere generar y no como el algoritmo anterior y siempre se empleará el mismo número de ciclos.

A continuación, presentamos una pequeña aplicación en la que se aplica este concepto de desordenar. Se trata de un programa que permita repartir cartas a un grupo de jugadores. El programa leerá de un archivo los DNI de los jugadores y le repartirá a cada uno cinco cartas; previamente a esto el programa barajará las cartas del mazo.

Para simular el mazo de cartas se definirá un arreglo con 52 elementos, que corresponde al número de cartas de un mazo convencional. Cada carta será identificada por un número consecutivo empezando de cero, luego para el proceso de impresión de una carta, el número se relacionará con la carta de la siguiente manera:

Las cartas que se encuentren entre 0 y 12 corresponderán a un palo de la baraja, las que se encuentren entre 13 y 25 a otro, entre 26 y 38 al tercero y de 39 a 51 al cuarto palo. De esta manera si recibimos un número cualquiera, bastará dividir el número entre 13 para obtener un valor 0, 1, 2 o 3 que podemos relacionarlo con un palo. Identificaremos los palos de las cartas con una letra C para los corazones♥, D para los diamantes♦, E para las espadas♠ y F para las flores♣.

Finalmente, el valor de la carta se obtiene del resto o módulo de la división.

Así por ejemplo si recibimos el número 38, se verá que corresponde al 12 de espadas♠, ya que 38 ÷ 13 da como resultado 2 ('C' + 2 = 'E' ➔ ♠) con residuo 12 (o sea la reina).

El programa se muestra a continuación:

```
#include <cstdlib>
#include <time.h>
#include "ManejoDeCartas.h"
#define NUM_CARTAS  52
#define MAX_JUGADORES 10

int main(int argc, char** argv) {
    int mazo[NUM_CARTAS], jugador[MAX_JUGADORES], numJug;

    srand(time(nullptr));
    leerJugadores(jugador, numJug);
    barajar(mazo);
    repartir(mazo, jugador, numJug);

    return 0;
}
```

```
#ifndef MANEJODECARTAS_H
#define MANEJODECARTAS_H

   void leerJugadores(int*,int&);
   void barajar(int*);
   void repartir(int*,int*,int);

#endif /* MANEJODECARTAS_H */
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include <cstdlib>
#include "ManejoDeCartas.h"
#define NUM_CARTAS  52

void leerJugadores(int *jugador, int &numJug){
    int dni;
    ifstream arch("Jugadores.txt",ios::in);
    if(not arch.is_open()){
        cout<<left<<"No se pudo abrir el archivo Jugadores.txt"<<endl;
        exit(1);
    }
    numJug = 0;
    while (1){
        arch>>dni;
        if(arch.eof()) break;
        jugador[numJug] = dni;
        numJug++;
    }
}

void barajar(int *mazo){
    int pos, aux, carta;
    for (int i=0; i<NUM_CARTAS; i++) mazo[i] = i;
    for (int limite=NUM_CARTAS -1 ;  limite>0; limite--){
        pos = rand()% NUM_CARTAS;
        aux = mazo[limite];
        mazo[limite] = mazo[pos];
        mazo[pos] = aux;
    }
}

void repartir(int *mazo, int *jugador, int numJug){
    int carta, pos = 0;
    char palo;
    for (int jug = 0; jug < numJug; jug++){
        cout<<right<<setw(10)<<jugador[jug]<<':';

        for (char c = 0; c < 5; c++){
            carta = mazo[pos] % 13 + 1;
            palo  = 'C' + mazo[pos]/13;
            cout<<" ["<<setw(2)<<carta<<palo<<']';
            pos++;
        }
        cout<<endl;
    }
}
```

El resultado es el siguiente:

```
  23651074:  [ 1 C] [ 5 E] [ 4 C] [13 D] [12 C]
  38763822:  [ 8 F] [ 3 D] [ 7 E] [ 3 C] [10 D]
  12345678:  [ 7 C] [ 2 C] [ 9 E] [ 6 D] [13 C]
  33559911:  [13 E] [10 F] [12 F] [ 1 D] [12 E]
  21436587:  [11 C] [ 6 C] [ 4 F] [ 8 E] [ 5 C]
```