Fundamentos de la programación

7 Algoritmos de ordenación

Grado en Ingeniería Informática
Grado en Ingeniería del Software
Grado en Ingeniería de Computadores

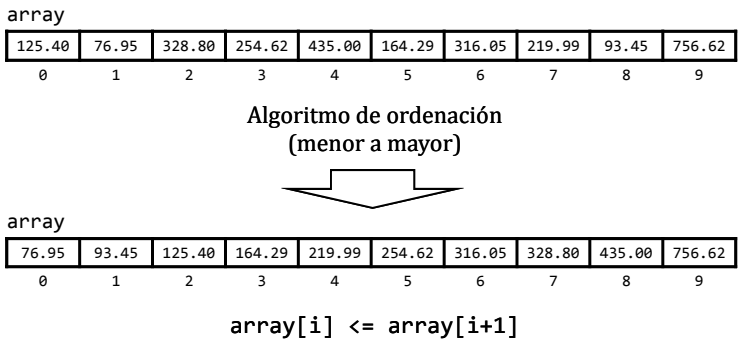
Facultad de Informática
Universidad Complutense

Índice

Luis Hernández Yáñez

Algoritmos de ordenación

Ordenación de listas



Tarea muy habitual: para mostrar los datos en orden o para facilitar las búsquedas.

Variadas formas de hacerlo (algoritmos), con mayor o menor rapidez.

Luis Hernández Yáñez

Algoritmos de ordenación

Ordenación de listas

Los datos de la lista deben poderse comparar entre sí (admitir los operadores relacionales).

Sentido de la ordenación:

- ✓ Ascendente (de menor a mayor)
- ✓ Descendente (de mayor a menor)

Algoritmos de ordenación básicos:

- ✓ Ordenación por *inserción*
- ✓ Ordenación por *selección directa*
- ✓ Ordenación por el *método de la burbuja*

Estos algoritmos se basan en comparaciones e intercambios.

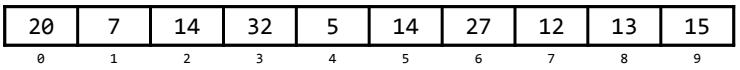
Hay otros algoritmos de ordenación mejores que se estudiarán en su momento.

Luis Hernández Yáñez

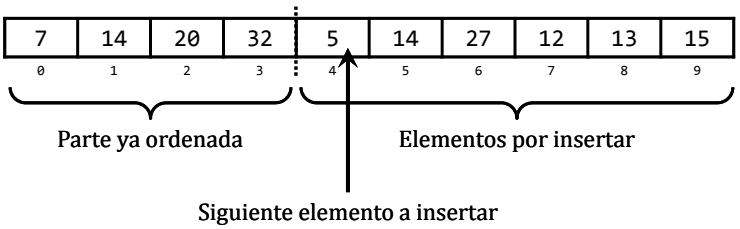
Ordenación por inserción

Ordenación de arrays por inserción

El array contiene inicialmente la lista desordenada:



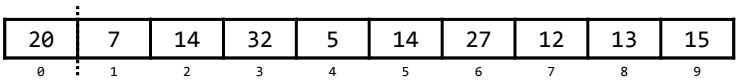
A medida que vayamos insertando los elementos en su lugar, una parte del array corresponderá a la lista ordenada y el resto a los elementos que todavía quedan por procesar.



Ordenación por inserción

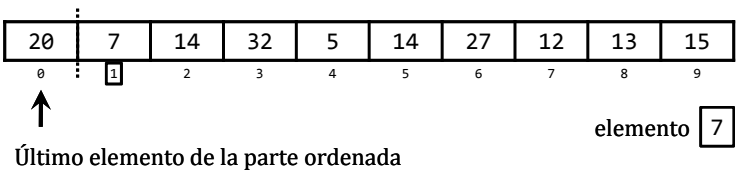
Ordenación de arrays por inserción

Situación inicial: Lista ordenada con un solo elemento. El primero del array:



Desde el segundo elemento hasta el último:

- *Insertamos el elemento en la parte ordenada*
- *Localizamos la posición del primer elemento estrictamente mayor…*

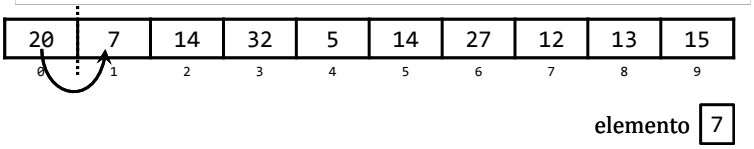


Ordenación por inserción

Ordenación de arrays por inserción

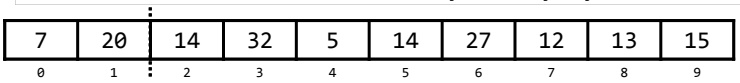
…

- *Desplazamos una posición a la derecha todos los estrictamente mayores.*



elemento 7

- *Y colocamos el elemento en curso en la posición que queda libre.*



elemento 7

Luis Hernández Yáñez

Ordenación por inserción

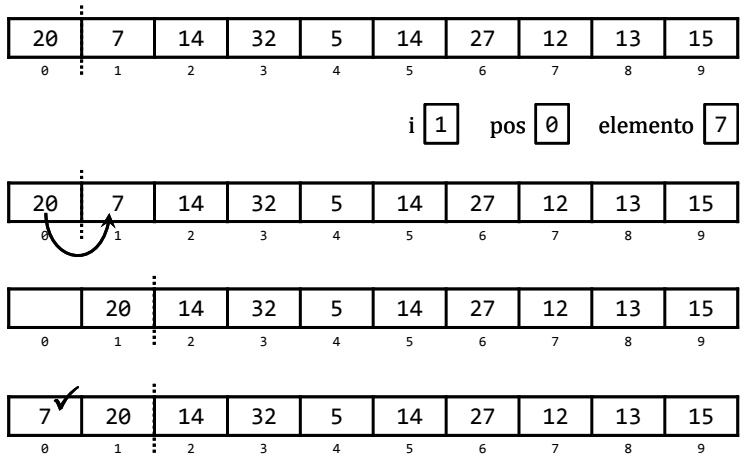
Ordenación de arrays por inserción

```
const int N = 10;
typedef double tLista[N];
tLista lista;
double elemento;
...
for (int i = 1; i < N; i++) {
// Desde el segundo elemento hasta el último
  elemento = lista[i];
  int pos = 0;
  bool enc= false;
  //búsqueda asimétrica
  while ((pos < i) && !enc){
    if(lista[pos] > elemento) enc= true;
    else pos++;
  }
  if(enc){ // pos --> primer elemento mayor
    for (int j = i; j > pos; j--) lista[j] = lista[j-1];
    lista[pos] = elemento;
  }
}
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

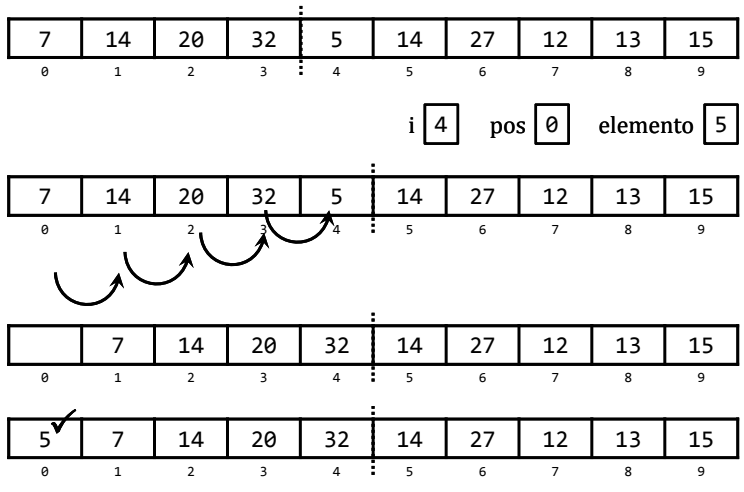
Ordenación de arrays por inserción



Luis Hernández Yáñez

Ordenación por inserción

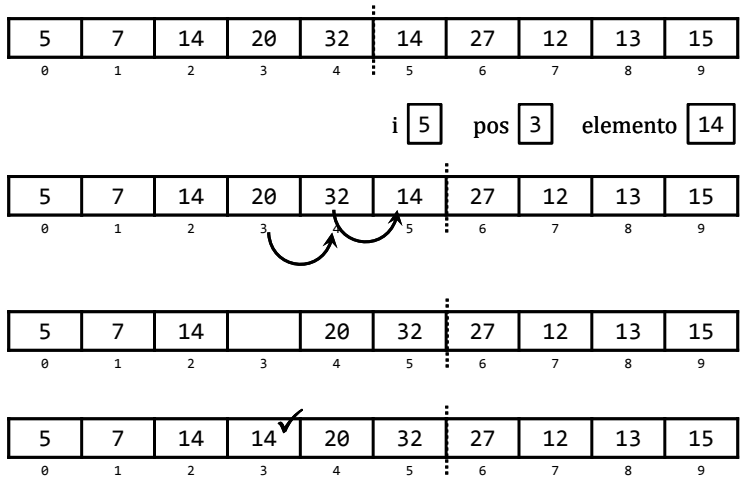
Ordenación de arrays por inserción



Luis Hernández Yáñez

Ordenación por inserción

Ordenación de arrays por inserción



Luis Hernández Yáñez

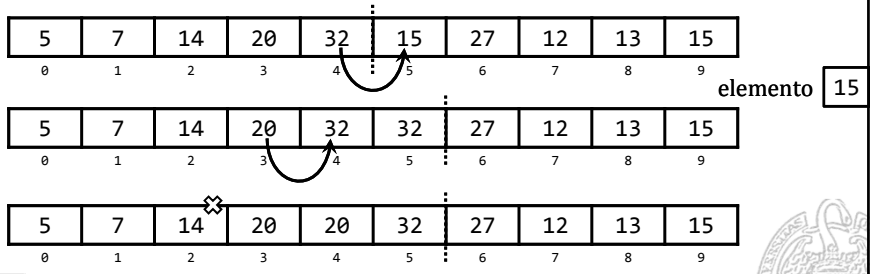
Ordenación por inserción

Ordenación de arrays por inserción (versión optimizada)

La inserción de cada elemento en su posición adecuada se puede llevar a cabo mediante una serie de comparaciones e intercambios:

Desde el segundo elemento hasta el último:

- *Obtener el elemento*
- *Buscamos el primer dato menor o igual, explorando la zona ya ordenada de derecha a izquierda, abriendo hueco al mismo tiempo*



Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Ordenación de arrays por inserción (versión optimizada)

```
const int N = 10;
typedef double tLista[N];
tLista lista;
...
for (int i = 1; i < N; i++) {
// Desde el segundo elemento hasta el último
  double elemento = lista[i];
  int pos = i-1; //i>=1
  bool enc = lista[pos] <= elemento;
  //esquema simétrico 2
  while ((pos > 0) && !enc) {
    lista[pos+1] = lista[pos];
    pos--;
    enc = lista[pos] <= elemento;
  }

  if (enc) lista[pos+1] = elemento;
  else {
    lista[1] = lista[0]; //desplazamos lista[0]
    lista[0] = elemento;
  }
}
```

Invariante:

Hemos desplazado `lista[pos+1..i-1]`

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Ordenación de arrays por inserción

```
#include <iostream>
#include <fstream>
using namespace std;
const int N = 10;
typedef double tLista[N];

void ordenarInsercion(tLista& lista);
void mostrar(const tLista lista);
bool cargar(tLista& lista);

int main() {
  tLista lista;
  if (!cargar(lista))
    cout << "Error de archivo: inexistente o con demasiados datos"
         << endl;
  else {
    cout << "Antes de ordenar:" << endl;
    mostrar(lista);
    ...
```

Luis Hernández Yáñez

Ordenación por inserción

```
    ...
    ordenarInsercion(lista);

    cout << "Después de ordenar:" << endl;
    mostrar(lista);
    cin.get();
  }
  return 0;
}

void ordenarInsercion(tLista& lista){
  int i, pos; double elemento; bool enc;
  for (i = 1; i < N; i++) {
    elemento = lista[i];
    pos = 0;
    enc= false;
    //búsqueda asimétrica
    while ((pos < i) && !enc){
      if(lista[pos] > elemento) enc= true;
      else pos++;
    }
    if(enc){ // pos --> primer elemento mayor
      for (int j = i; j > pos; j--) lista[j] = lista[j-1];
      lista[pos] = elemento;
    }
  }
}
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Claves de ordenación

A menudo las listas que hay que ordenar no son simplemente arrays de datos simples, sino arrays de estructuras.

```
const int N = 100;
typedef struct {
  int codigo;
  string nombre;
  double sueldo;
} tDato;
typedef tDato tLista[N];
tLista lista;
```

```
tDato elemento;
```

Clave de ordenación: Campo en el que se basan las comparaciones.

- En el método de inserción "estándar"

```
if(lista[pos].nombre > elemento.nombre) enc= true;
else pos++;
```

- En el método de inserción optimizado

```
enc = lista[pos].nombre <= elemento.nombre;
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Claves de ordenación

Podemos crear una función para el operador relacional:

```
bool operator>(tDato opIzq, tDato opDer) {
  return opIzq.nombre > opDer.nombre;
}
```

```
tDato elemento;
```

- En el método de inserción "estándar"

```
if(lista[pos] > elemento) enc= true;
else pos++;
```

- En el método de inserción optimizado

```
enc = !(lista[pos] > elemento); //a≤b ↔ !(a>b)
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Claves de ordenación

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
#include <iomanip>
const int N = 100;
typedef struct {
  int codigo;
  string nombre;
  double sueldo;
} tDato;
typedef struct {
  tDato datos[N];
  int cont;
} tLista;

void ordenarInsercion(tLista& lista);
bool operator>(const tDato& opIzq, const tDato& opDer);
void mostrar(const tLista& lista);
void cargar(ifstream& ent, tLista& lista);
```

Luis Hernández Yáñez

Ordenación por inserción

```
int main() {
  tLista lista;
  lista.cont = 0;
  ifstream archivo;
  archivo.open("datos.txt");
  if (!archivo.is_open()) {
    cout << "Error de apertura del archivo" << endl;
    return 1;
  }
  cargar(archivo, lista);
  cout << "Antes de ordenar:" << endl;
  mostrar(lista);

  ordenarInsercion(lista);

  cout << "Despues de ordenar:" << endl;
  mostrar(lista);

  cin.get();
  return 0;
}
```

Luis Hernández Yáñez

Ordenación por inserción

```
void ordenarInsercion(tLista& lista){
  int i, pos; tDato elemento; bool enc;
  for (i = 1; i < lista.cont; i++) {
    elemento = lista.datos[i];
    pos = 0;
    enc= false;
    //búsqueda asimétrica
    while ((pos < i) && !enc){
      if(lista.datos[pos] > elemento) enc= true;
      else pos++;
    }
    if(enc){ // pos --> primer elemento mayor
      for (int j = i; j > pos; j--)
            lista.datos[j] = lista.datos[j-1];
      lista.datos[pos] = elemento;
    }
  }
}

bool operator>(const tDato& opIzq, const tDato& opDer){
  return opIzq.nombre > opDer.nombre;
}
```

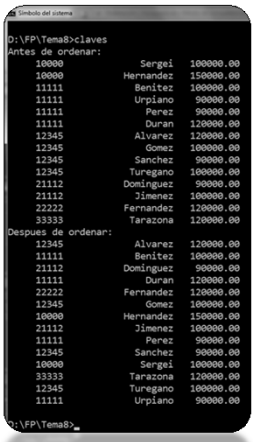
Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

```
...

void mostrar(const tLista& lista) {
  for (int i = 0; i < lista.cont; i++) {
    cout << setw(10) << lista.datos[i].codigo
         << setw(20) << lista.datos[i].nombre
         << setw(12) << fixed << setprecision(2)
         << lista.datos[i].sueldo << endl;
  }
}

void cargar(ifstream& ent, tLista& lista){
  int cod; string nom; double suel;
  while ((lista.cont < N) && (archivo >> cod)
      && (archivo >> nom) && (archivo >> suel)){
      lista.datos[lista.cont].codigo = cod;
      lista.datos[lista.cont].nombre = nom;
      lista.datos[lista.cont].sueldo = suel;
      lista.cont++;
  }
}
```



Luis Hernández Yáñez

Ordenación por inserción

Estabilidad de la ordenación

Un algoritmo de ordenación es estable si al tener una lista ordenada por una clave y ordenarla por otra clave distinta, dentro de esta segunda ordenación se preserva el orden de la primera.

Supongamos que tenemos una lista con la estructura anterior, en donde cualquiera de los campos puede ser clave de ordenación (`Codigo`, `Nombre`, `Sueldo`).

Supongamos ahora que hemos ordenado alfabéticamente la lista (por el campo `Nombre`) y que ha quedado así:

12345	Alvarez	120000
11111	Benitez	100000
21112	Dominguez	90000
11111	Duran	120000
22222	Fernandez	120000
12345	Gomez	100000
10000	Hernandez	150000
21112	Jimenez	100000
11111	Perez	90000
12345	Sanchez	90000
10000	Sergei	100000
33333	Tarazona	120000
12345	Turegano	100000
11111	Urpiano	90000

Luis Hernández Yáñez

Ordenación por inserción

Estabilidad de la ordenación

Si ahora ordenamos por el campo `Codigo`:

10000	Sergei	100000
10000	Hernández	150000
11111	Benítez	100000
11111	Urpiano	90000
11111	Pérez	90000
11111	Durán	120000
12345	Alvarez	120000
12345	Gómez	100000
12345	Sánchez	90000
12345	Turégano	100000
21112	Domínguez	90000
21112	Jiménez	100000
22222	Fernández	120000
33333	Tarazona	120000

No estable:
Los nombres no mantienen sus posiciones relativas.

10000	Hernández	150000
10000	Sergei	100000
11111	Benítez	100000
11111	Durán	120000
11111	Pérez	90000
11111	Urpiano	90000
12345	Alvarez	120000
12345	Gómez	100000
12345	Sánchez	90000
12345	Turégano	100000
21112	Domínguez	90000
21112	Jiménez	100000
22222	Fernández	120000
33333	Tarazona	120000

Estable:
Los nombres mantienen sus posiciones relativas.

Luis Hernández Yáñez

Ordenación por inserción

Estabilidad de la ordenación

Los dos métodos de ordenación por inserción que hemos visto son estables.

No lo son serían si usáramos las siguientes comparaciones:

```
                tDato elemento;
bool operator<(const tDato& a, const tDato& b);
bool operator>=(const tDato& a, const tDato& b);
```

- En el método de inserción "estándar"

```
if(lista[pos] >= elemento) enc= true;
else pos++;
```

- En el método de inserción optimizado

```
enc = lista[pos] < elemento;
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Casos de estudio

- ✓ Lista inicialmente ordenada.
- ✓ Lista inicialmente inversamente ordenada.
- ✓ Lista con disposición aleatoria

¿Trabaja menos, más o igual la ordenación en cada caso?

Si el algoritmo de ordenación trabaja menos cuanto *más ordenada* está inicialmente la lista, se dice que la ordenación es *natural*.

Ordenación por inserción con la lista inicialmente ordenada:

- ✓ Versión estándar:
  Se ahorra los desplazamientos, pero hace el máximo número de comparaciones. No natural.
- ✓ Versión optimizada:
  Trabaja mucho menos, ya que basta una comparación cada vez. Natural.

Ordenación por inserción

Complejidad y eficiencia

Mide la cantidad de recursos en tiempo (y espacio) que necesita un algoritmo para resolver un problema.

La complejidad algorítmica no proporcionan medidas absolutas sino medidas relativas al tamaño del problema.

El tiempo (medido en pasos) requerido por un algoritmo en función del tamaño de los datos: T(N)

Contaremos las operaciones que realiza el algoritmo en base a la dimensión de la lista a ordenar.

En este caso:

- ✓ Comparaciones
- ✓ Asignaciones en componentes del array

Asumimos que ambas requieren un tiempo similar.

Ordenación por inserción

Complejidad y eficiencia

Ordenación por inserción optimizada:

```
...
for (int i = 1; i < N; i++) {
  double elemento = lista[i];
  int pos = i-1; //i>=1
  bool enc = lista[pos] <= elemento;

  //esquema simétrico 1
  while ((pos > 0) && !enc) {

    lista[pos+1] = lista[pos];
    pos--;
    enc = lista[pos] <= elemento;
  }

  if (enc) lista[pos+1] = elemento;
  else {
    lista[1] = lista[0]; //desplazamos lista[0]
    lista[0] = elemento;
  }
}
```

Comparación

Asignación

Comparación

Asignación

Asignaciones

El número de veces que se ejecutan estas asignaciones y comparaciones depende del bucle en el que se encuentren.

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Complejidad y eficiencia

Ordenación por inserción optimizada:

```
...
...
for (int i = 1; i < N; i++) {
  double elemento = lista[i];
  int pos = i-1; //i>=1
  bool enc = lista[pos] <= elemento;

  //esquema simétrico 1
  while ((pos > 0) && !enc) {

    lista[pos+1] = lista[pos];
    pos--;
    enc = lista[pos] <= elemento;
  }

  if (enc) lista[pos+1] = elemento;
  else {
    lista[1] = lista[0]; //desplazamos lista[0]
    lista[0] = elemento;
  }
}
```

N - 1 ciclos

Nº variable de ciclos

Casos en los que el `while` interior se ejecuta más (*caso peor*) y casos en los que se ejecuta menos (*caso mejor*).

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Complejidad y eficiencia

Ordenación por inserción optimizada:

- ✓ Caso mejor: lista inicialmente ordenada
  La primera comparación falla y no se realiza ningún intercambio.
  T(N)= (N-1) * (1 comparación + 1 asignación)
- ✓ Caso peor: lista inicialmente ordenada al revés
  Para cada pos, entre i-1 y 1, se realiza 1 asignación y 1comparación.
  Además 1 comparación antes y 2 asignaciones después

$$T(N) = \sum_{i=1}^{N-1} (2(i-1)+3) = 2\sum_{i=1}^{N-1} i + (N-1)$$

$$= 2\frac{N(N-1)}{2} + (N-1) = (N-1)(N+1)$$

$$= N^2 - 1$$

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por inserción

Complejidad y eficiencia

Ordenación por inserción (optimizada):

- ✓ Caso mejor: T(N)= 2N - 2
  Se dice que tiene complejidad lineal
  $T(N) \in O(N)$
- ✓ Caso peor: $T(N)= N^2 - 1$
  Se dice que tiene complejidad cuadrática
  $T(N) \in O(N^2)$
- ✓ Caso medio (distribución aleatoria de los elementos)
  $T(N) \in O(N^2)$

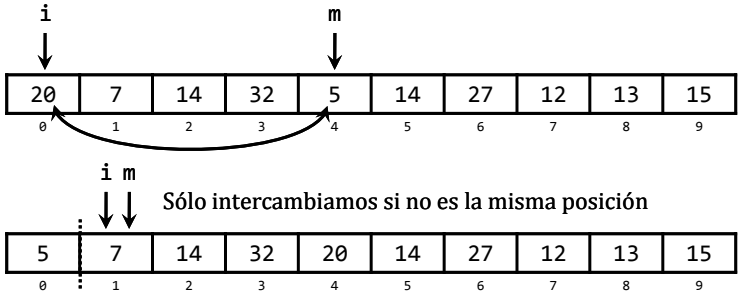
Hay algoritmos de ordenación mejores!

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ordenación por selección directa

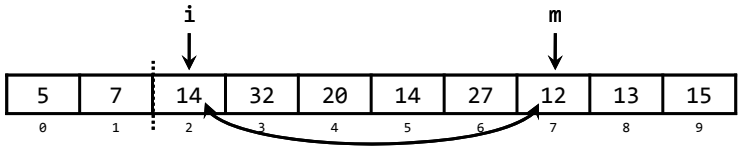
Algoritmo de ordenación por selección directa

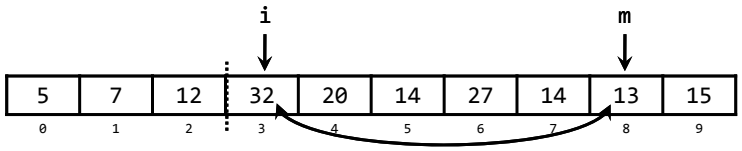
Desde la primera posición, `i` = 0, hasta la penúltima (N-2):
- *Calcular `m`: posición con el menor elemento en [`i`..(N-1)]*
- *Intercambiar el elemento de la posición `i` con el de la posición `m`*

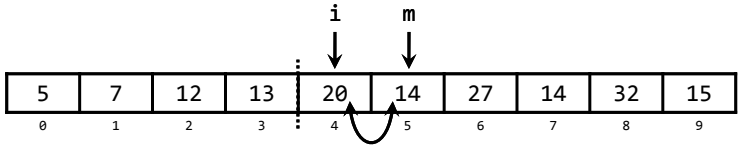


Ordenación por selección directa

Algoritmo de ordenación por selección directa







Ordenación por selección directa

Algoritmo de ordenación por selección directa

```
const int N = 10;
typedef double tLista[N];
tLista lista;
double tmp;
...
for (int i = 0; i < N - 1; i++) {
// Desde el primer elemento hasta el penúltimo
  int menor = i;
  // Recorrido
  for (int j = i + 1; j < N; j++)
  // Desde i+1 hasta el final
    if (lista[j] < lista[menor]) menor = j;

  //if (menor > i){
  tmp = lista[i];
  lista[i] = lista[menor];
  lista[menor] = tmp;
  //}
}
```

Luis Hernández Yáñez

Ordenación por selección directa

Algoritmo de ordenación por selección directa

¿Cuántas comparaciones se llevan a cabo?

El bucle externo se ejecuta N-1 veces y dentro de él se llevan a cabo tantas comparaciones como elementos queden en la lista. Además se ejecutan 2 asignaciones en posiciones del array:

$$T(N) = \sum_{i=0}^{N-2}((N-i-1)+2) = \sum_{i=0}^{N-2}(N-i+1) = \sum_{i=0}^{N-2} i$$

$$= (N+1)(N-1) - \frac{(N-1)(N-2)}{2}$$

$$= \frac{(N-1)(N+4)}{2}$$

Mismo número de comparaciones en todos los casos

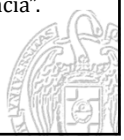
Complejidad cuadrática

$T(N) \in O(N^2)$

No es estable: los intercambios tras el bucle interno se realizan "a larga distancia".

No es natural (trabaja siempre igual).

Luis Hernández Yáñez
Pedro J. Martín de la Calle

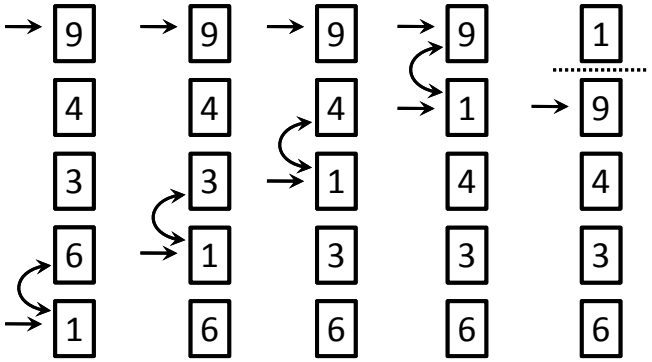


Método de la burbuja

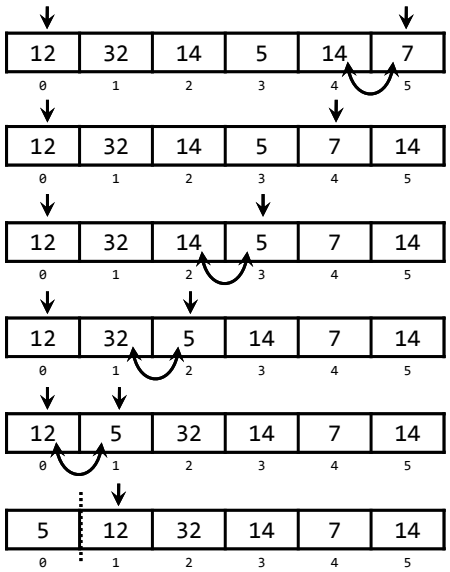
Algoritmo de ordenación por el método de la burbuja

Es una variación del método de selección directa.

El elemento menor va subiendo hasta alcanzar su posición definitiva, como una burbuja en el agua sube hacia la superficie.



Método de la burbuja



Método de la burbuja

Algoritmo de ordenación por el método de la burbuja

Desde la primera posición, `i` = 0, hasta la penúltima (N-2):
Desde la última posición , `j` = N-1, hasta `i+1` hacer:
Si el elemento en `j` es menor que el elemento en `j-1`, intercambiarlos

```
const int N = 10;
typedef double tLista[N];
tLista lista;
double tmp;
...
for (int i = 0; i < N - 1; i++) {
// Desde el primer elemento hasta el penúltimo
  for (int j = N - 1; j > i; j--)
  // Desde el último hasta el siguiente a i
    if (lista[j] < lista[j-1]) {
      tmp = lista[j];
      lista[j] = lista[j-1];
      lista[j-1] = tmp;
    }
```


Método de la burbuja

Algoritmo de ordenación por el método de la burbuja

Complejidad cuadrática

$T(N) \in O(N^2)$

No tiene un comportamiento natural.
Sí es estable (mantiene el orden relativo).

Una mejora:

Si en una iteración del bucle exterior no se ha realizado ningún intercambio, entonces la lista está ordenada y no es necesario seguir.

14	14	14	12
16	16	12	14
35	12	16	16
12	35	35	35
50	50	50	50

La lista ya está ordenada.
No hace falta seguir.
En el siguiente paso
no hay intercambios.

Método de la burbuja

Método de la burbuja mejorado

```
const int N = 10;
typedef double tLista[N];
tLista lista;
bool subirBurbuja( int i, tLista& lista);
// Sube la burbuja en lista[i+1..N-1]
// i<N-1
// Devuelve si ha habido intercambios

...
// Buscar una pasada sin intercambios
bool encontrado = false;
int i = 0;
// Esquema asimétrico
while ((i < N - 1) && !encontrado) {
   if !subirBurbuja(i, lista) encontrado= true;
   else i++;
}
```

Esta variación SÍ tiene un comportamiento natural.

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Método de la burbuja

Método de la burbuja mejorado

```
bool subirBurbuja( int i, tLista& lista) {

  double tmp;
  bool intercambios= false;

  for (int j = N - 1; j > i; j--) {
  // Desde el último hasta i+1
    if (lista[j] < lista[j-1]) {
      tmp = lista[j];
      lista[j] = lista[j-1];
      lista[j-1] = tmp;
      intercambios = true;
    }
  }
  return intercambios;
}
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Búsquedas en listas ordenadas

Búsqueda secuencial

En las secuencias no ordenadas buscamos recorriendo hasta que encontremos el elemento o lleguemos al final.

Si la secuencia o lista está ordenada podemos parar antes de llegar al final en el caso de que el elemento no esté.

5	7	12	13	14	14	15	20	27	32
0	1	2	3	4	5	6	7	8	9

Si, por ejemplo, buscamos el 17, al llegar al 20 ya sabremos que no está, pues a continuación todos los valores son >= 20 y, por lo tanto > 17.

Consecuencia: se trata de buscar el primer elemento de la lista que sea mayor o igual que el dato que se busca.

Luis Hernández Yáñez

Búsquedas en listas ordenadas

Búsqueda secuencial

```
const int N = 10;
typedef double tLista[N];
tLista lista; // lista ordenada
double buscado;

int i = 0;
bool encontrado = false;

// Búsqueda asimétrica
while ((i < N) && !encontrado) {
  if( lista[i] >= buscado) encontrado = true;
  else i++;
}

// CASOS:
// encontrado && (lista[i] == buscado): Encontrado en la posición i
// encontrado && (lista[i] > buscado):  No encontrado,
//                                debería estar justo antes que lista[i]
// !encontrado: No encontrado, debería estar detrás de todos

if(encontrado && (lista[i] == buscado)) { // Encontrado!!
  ...
}
```

Complejidad lineal
O(N)

Invariante:
`lista[0 … i-1] < buscado`

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Búsquedas en listas ordenadas

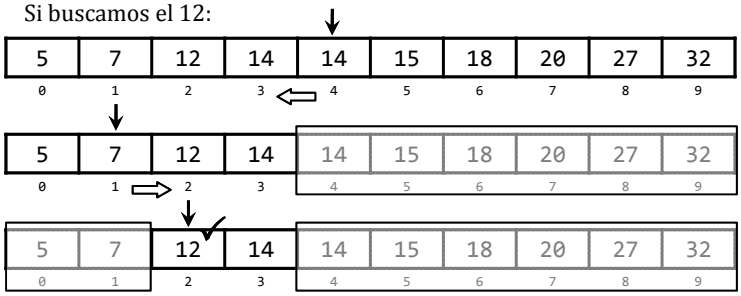
Búsqueda binaria

Una forma mucho más rápida de buscar que aprovecha la ordenación.

Comparamos con el valor que esté en el medio de la lista.
- *Si es el que buscamos, terminamos.*
- *Si no, buscamos en la primera mitad de la lista si ese elemento es mayor que el que buscamos y en la segunda mitad en caso contrario.*

Repetimos el proceso hasta encontrarlo o quedarnos sin sublista.

Si buscamos el 12:



Luis Hernández Yáñez

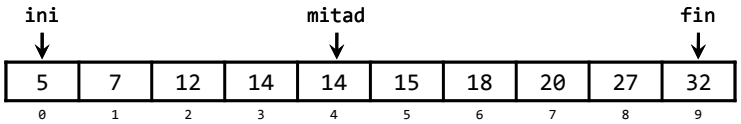
Búsquedas en listas ordenadas

Búsqueda binaria

Vamos buscando en sublistas cada vez más pequeñas.

Debemos limitar el segmento de la lista donde estamos buscando.

Inicialmente tenemos toda la lista:



El índice del elemento en la mitad es: `mitad = (ini + fin) / 2`

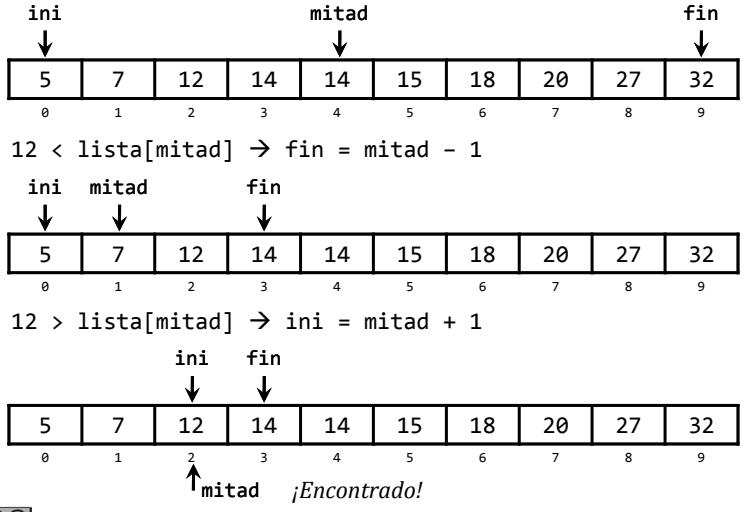
Si no se ha encontrado, ¿dónde hay que buscar a continuación?

- Si el buscado es menor que el que está en la mitad: `fin = mitad - 1`
- Si el buscado es mayor que el que está en la mitad: `ini = mitad + 1`

Luis Hernández Yáñez

Búsquedas en listas ordenadas

Búsqueda binaria Buscamos el 12

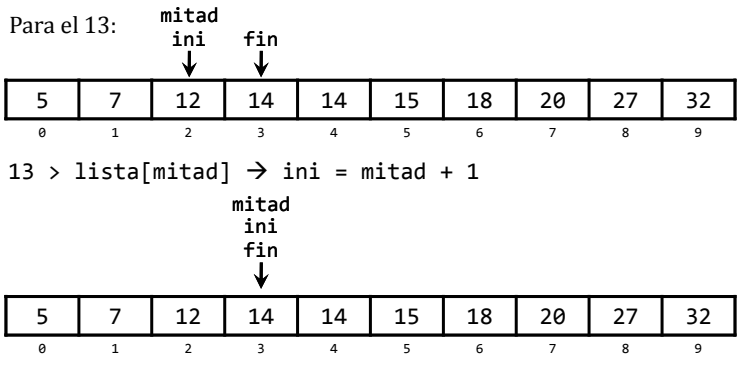


Búsquedas en listas ordenadas

Búsqueda binaria

Si el elemento no está, acabaremos quedándonos sin sublista: `ini > fin`

Para el 13:



`13 < lista[mitad]` → `fin = mitad - 1`

¡¡¡ `ini > fin` !!! No hay ya dónde seguir buscando → No está.

Búsquedas en listas ordenadas

Búsqueda binaria

```
const int N = 10;
typedef double tLista[N];
tLista lista;
...
double buscado;
cout << "Valor a buscar: "; cin >> buscado;
int ini = 0, fin = N – 1, mitad;
bool encontrado = false;
while ((ini <= fin) && !encontrado) {
  mitad = (ini + fin) / 2; // División entera
  if (buscado == lista[mitad]) encontrado = true;
  else if (buscado < lista[mitad]) fin = mitad - 1;
  else ini = mitad + 1;
}
if (encontrado)
  cout << "Encontrado en la posición " << mitad + 1 << endl;
else cout << "No encontrado!" << endl;
```


Búsquedas en listas ordenadas

Búsqueda binaria

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

const int N = 100;
typedef int tArray[N];
typedef struct {
  tArray elementos;
  int cont;
} tLista;

bool buscar(tLista lista, int buscado, int& pos);

int main() {
  tLista lista;
  lista.cont = 0;
  ifstream archivo;
  archivo.open("enteros.txt"); // Existe y es correcto
                                                    ...
```


Búsquedas en listas ordenadas

```
...
  int dato;
  while (archivo >> dato) {
    lista.elementos[lista.cont] = dato;
    lista.cont++;
  }
  for (int i = 0; i < lista.cont; i++)
    cout << lista.elementos[i] << "  ";
  cout << endl;
  int buscado, pos;
  cout << "Valor a buscar: "; cin >> buscado;
  if (buscar(lista, buscado, pos))
    cout << "Encontrado en la posición " << pos + 1 << endl;
  else
    cout << "No encontrado!" << endl;

  return 0;
}
                                                          ...
```

Luis Hernández Yáñez

Búsquedas en listas ordenadas

```
...

bool buscar(tLista lista, int buscado, int& pos) {
  int ini = 0, fin = lista.cont - 1, mitad;
  bool encontrado = false;
  while ((ini <= fin) && !encontrado) {
    mitad = (ini + fin) / 2; // División entera
    if (buscado == lista.elementos[mitad]) encontrado = true;
    else if (buscado < lista.elementos[mitad]) fin = mitad - 1;
    else ini = mitad + 1;
  }
  if (encontrado) pos = mitad;

  return encontrado;
}
```

Luis Hernández Yáñez

Búsquedas en listas ordenadas

Búsqueda binaria

¿Qué orden de complejidad tiene?

Caso peor:

El elemento no está en la lista o se encuentra en sublista de 1 elemento.

Nº de comparaciones = Nº de vueltas que se ejecuta el while

= Nº de veces que visitamos el índice mitad para partir la sublista

$$N \to {}^{N}/_{2} \to {}^{N}/_{4} \to {}^{N}/_{8} \to \ \dots \to 4 \to 2 \to 1 \to \begin{cases} 0 \text{ (Fallo!)} \\ \text{Éxito!} \end{cases}$$

El número de vueltas en el peor caso es el número de flechas →

Si hacemos que N sea igual a 2^k, esa serie queda como:

$$2^k \to 2^{k-1} \to 2^{k-2} \to 2^{k-3} \to \ \dots \to 2^2 \to 2^1 \to 2^0 \to \begin{cases} 0 \text{ (Fallo!)} \\ \text{Éxito!} \end{cases}$$

En total k +1 vueltas:

$$N = 2^k \rightarrow T(N) = 1 + \log_2 N$$

Complejidad logarítmica: $T(N) \in O(\log_2 N)$

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Búsquedas en listas ordenadas

Órdenes de complejidad

$O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N^3)$ ...

N	$\log_2$ N	N^2
1	0	1
2	1	4
4	2	16
8	3	64
16	4	256
32	5	1024
64	6	4096
128	7	16384
256	8	65536
...		

Luis Hernández Yáñez

Operaciones con listas ordenadas

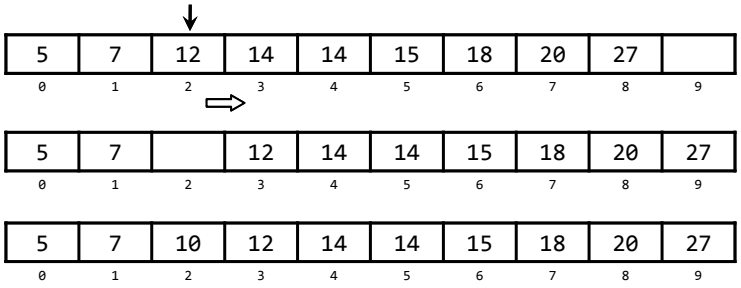
Inserciones y eliminaciones en listas ordenadas

Hay que insertar de forma que la lista siga ordenada:

- Buscar el lugar, desplazar los siguientes a la derecha y colocar.

Al eliminar no hay que dejar huecos.

Por ejemplo, insertamos el 10:



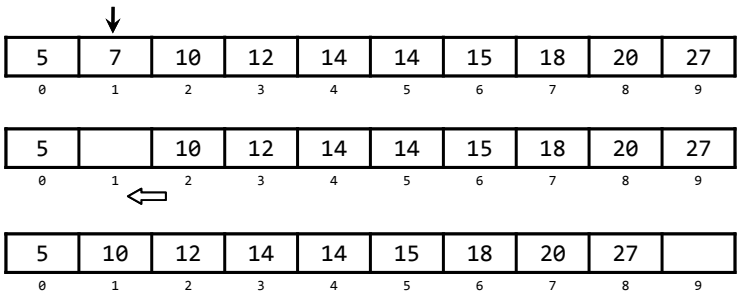
¡Error si la lista está llena!

Luis Hernández Yáñez

Operaciones con listas ordenadas

Inserciones y eliminaciones en listas ordenadas

Por ejemplo, eliminamos el 7 (el 2º):



¡Error si la posición no es válida!

Luis Hernández Yáñez

Operaciones con listas ordenadas

Gestión de una tabla de datos ordenada

`tabla.cpp`

Te proporcionamos un ejemplo bastante completo de gestión de una tabla de datos ordenada por un campo `nombre`:

```
const int N = 100;
typedef struct {
  int codigo;
  string nombre;
  double sueldo;
} tRegistro;

typedef tRegistro tLista[N];

typedef struct {
  tLista registros;
  int cont;
} tTabla;
```


Operaciones con listas ordenadas

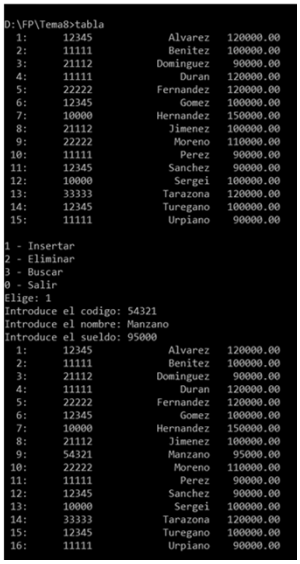
Gestión de una tabla de datos ordenada

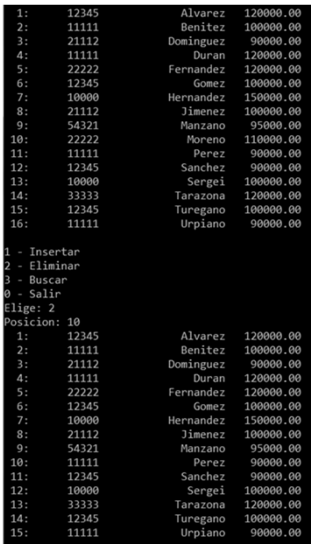
Incluye estos subprogramas:

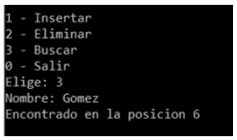
```
void mostrarDato(int pos, tRegistro registro);
// Línea con los datos del registro precedidos de su posición
void mostrar(const tTabla& tabla);
bool operator>(tRegistro opIzq, tRegistro opDer);
bool operator<(tRegistro opIzq, tRegistro opDer);
tRegistro nuevo(); // Lee del teclado los datos de un registro
bool insertar(tTabla& tabla, tRegistro registro);
bool eliminar(tTabla& tabla, int pos); // pos = 1..N
int buscar(tTabla tabla, string nombre); // -1 si no está
bool cargar(tTabla& tabla);
void guardar(tTabla tabla);
int menu(); // insertar, eliminar o buscar
```

Para el usuario las posiciones de los registros en la tabla comienzan en 1.

Operaciones con listas ordenadas







Luis Hernández Yáñez

Mezcla de listas ordenadas

Mezcla de dos listas ordenadas en arrays

```
const int N = 100;
typedef struct {
  int elementos[N];
  int cont;
} tLista;
```

Un índice para cada lista, inicializados a 0 (principio de las listas).

Mientras que no lleguemos al final de alguna de las dos listas:

- *Elegimos el elemento menor de los que tienen esos índices.*
- *Lo copiamos en la lista resultado y avanzamos ese índice una posición.*

Copiamos los que queden en la lista que no se haya acabado en la lista resultado.

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Mezcla de listas ordenadas

Mezcla de dos listas ordenadas en arrays

```
void mezcla(tLista lista1, tLista lista2, tLista& listaM) {
  int pos1 = 0, pos2 = 0;
  int cont = 0;
  bool enc = (cont==N);

  // buscamos el N-ésimo dato introducido en listaM
  while ((pos1 < lista1.cont) && (pos2 < lista2.cont) && !enc) {
    if (lista1.elementos[pos1] < lista2.elementos[pos2]) {
      listaM.elementos[cont] = lista1.elementos[pos1];
      pos1++;
    }
    else {
      listaM.elementos[cont] = lista2.elementos[pos2];
      pos2++;
    }
    cont++;
    enc = (cont==N);
  }
  ...
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Mezcla de listas ordenadas

Mezcla de dos listas ordenadas en arrays

```
...
  // Pueden quedar datos en alguna de las listas
  if (pos1 < lista1.cont)
    while ((pos1 < lista1.cont) && !enc) {
      listaM.elementos[cont] = lista1.elementos[pos1];
      pos1++;
      cont++;
      enc = (cont==N);
    }
  else // pos2 < lista2.cont
    while ((pos2 < lista2.cont) && !enc) {
      listaM.elementos[cont] = lista2.elementos[pos2];
      pos2++;
      cont++;
      enc = (cont==N);
    }
  listaM.cont = cont;
}
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Mezcla de listas ordenadas

Mezcla de dos listas ordenadas en archivos

```
bool mezcla(string nombre1, string nombre2, string nombreM) {
// Mezcla las secuencias en los archivos nombre1 y nombre2
// generando la secuencia mezclada en el archivo nombreM
  ifstream archivo1, archivo2;
  ofstream mezcla;
  bool ok = true;
  archivo1.open(nombre1.c_str());
  if (!archivo1.is_open())
    ok = false;
  else {
    archivo2.open(nombre2.c_str());
    if (!archivo2.is_open())
      ok = false;
    else { // Ambos archivos existen
      mezcla.open(nombreM.c_str());
      int dato1, dato2;
      bool eof1, eof2;
      eof1 = !(archivo1 >> dato1);
      eof2 = !(archivo2 >> dato2);
      ...
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Mezcla de listas ordenadas

Mezcla de dos listas ordenadas en archivos

```
...   while (!eof1 && !eof2) { // Recorrido:
        if (dato1 < dato2) {   // Mientras quede algo en ambos archivos
          mezcla << dato1 << endl;
          eof1 = !(archivo1 >> dato1);
        } else {
          mezcla << dato2 << endl;
          eof2 = !(archivo2 >> dato2);
        }
      } // Uno de los dos archivos se ha acabado
      if (!eof1) {
        mezcla << dato1 << endl; // Quedaba uno por guardar!
        while (archivo1 >> dato1) mezcla << dato1 << endl; // Recorrido
      } else {
        mezcla << dato2 << endl; // Quedaba uno por guardar!
        while (archivo2 >> dato2) mezcla << dato2 << endl; // Recorrido
      }
      archivo2.close();
      mezcla.close();
    }
    archivo1.close();
  }
  return ok;
}
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Referencias bibliográficas

- *Programación en C++ para ingenieros*
  F. Xhafa et al. Thomson, 2006
- *El lenguaje de programación C++* (Edición especial)
  B. Stroustrup. Addison-Wesley, 2002

Luis Hernández Yáñez

Acerca de *Creative Commons*

Licencia CC (Creative Commons)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

- Reconocimiento (*Attribution*):
  En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
- No comercial (*Non commercial*):
  La explotación de la obra queda limitada a usos no comerciales.
- Compartir igual (*Share alike*):
  La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.

Luis Hernández Yáñez