

7

Algoritmos de ordenación

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

Luis Hernández Yáñez
Facultad de Informática
Universidad Complutense



Índice

Algoritmos de ordenación	651
Algoritmo de ordenación por inserción	654
Ordenación de arrays por inserción	665
Algoritmo de ordenación por inserción con intercambios	672
Claves de ordenación	680
Estabilidad de la ordenación	688
Complejidad y eficiencia	692
Ordenaciones naturales	694
Ordenación por selección directa	701
Método de la burbuja	716
Listas ordenadas	722
Búsquedas en listas ordenadas	729
Búsqueda binaria	731



Fundamentos de la programación

Algoritmos de ordenación



Algoritmos de ordenación

Ordenación de listas

array

125.40	76.95	328.80	254.62	435.00	164.29	316.05	219.99	93.45	756.62
0	1	2	3	4	5	6	7	8	9

Algoritmo de ordenación
(de menor a mayor)



array

76.95	93.45	125.40	164.29	219.99	254.62	316.05	328.80	435.00	756.62
0	1	2	3	4	5	6	7	8	9

$\text{array}[i] \leq \text{array}[i + 1]$

Mostrar los datos en orden, facilitar las búsquedas, ...

Variadas formas de hacerlo (algoritmos)



Algoritmos de ordenación

Ordenación de listas

Los datos de la lista deben poderse comparar entre sí

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*

Los algoritmos se basan en comparaciones e intercambios

Hay otros algoritmos de ordenación mejores



Algoritmo de ordenación por inserción

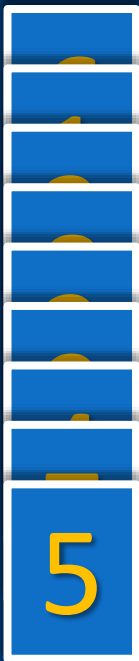


Ordenación por inserción

Algoritmo de ordenación por inserción

Partimos de una lista vacía

Vamos insertando cada elemento en el lugar que le corresponda



Baraja de nueve cartas numeradas del 1 al 9

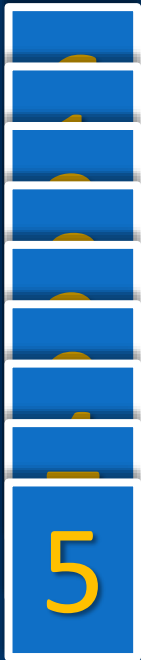
Las cartas están desordenadas

Ordenaremos de menor a mayor (ascendente)



Ordenación por inserción

Algoritmo de ordenación por inserción



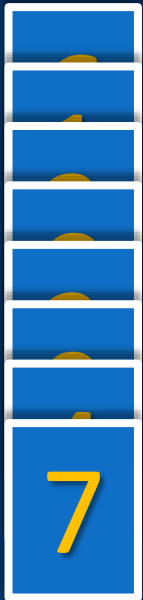
Colocamos el primer elemento en la lista vacía

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



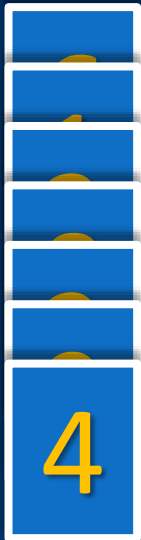
El 7 es mayor que todos los elementos de la lista
Lo insertamos al final

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



Primer elemento (5) mayor que el nuevo (4):
Desplazamos todos una posición a la derecha
Insertamos el nuevo en la primera posición

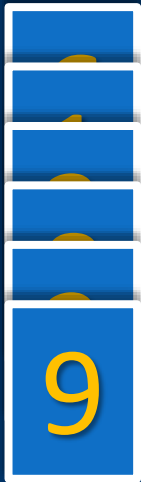
Hemos insertado el elemento en su lugar

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



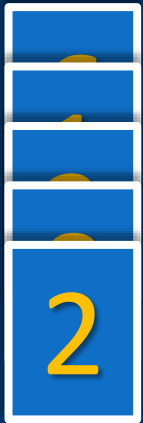
9 es mayor que todos los elementos de la lista
Lo insertamos al final

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



Primer elemento (4) mayor que el nuevo (2):
Desplazamos todos una posición a la derecha
Insertamos el nuevo en la primera posición

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



El 9 es el primer elemento mayor que el nuevo (8):

Desplazamos desde ese hacia la derecha

Insertamos donde estaba el 9

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



Segundo elemento (4) mayor que el nuevo (3):
Desplazamos desde ese hacia la derecha
Insertamos donde estaba el 4

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción



Primer elemento (2) mayor que el nuevo (1):
Desplazamos todos una posición a la derecha
Insertamos el nuevo en la primera posición

Lista ordenada:



Ordenación por inserción

Algoritmo de ordenación por inserción

6

El 7 es el primer elemento mayor que el nuevo (6):
Desplazamos desde ese hacia la derecha
Insertamos donde estaba el 7

!!! LISTA ORDENADA !!!

Lista ordenada:



Ordenación por inserción

Ordenación de arrays por inserción

El array contiene inicialmente la lista desordenada:

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

A medida que insertamos: dos zonas en el array

Parte ya ordenada y elementos por procesar

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

Parte ya ordenada

Elementos por insertar

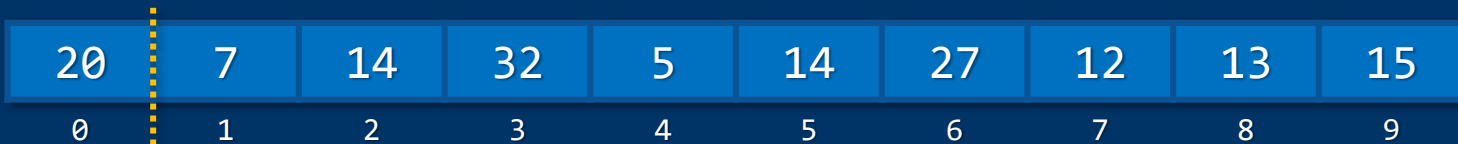
Siguiente elemento a insertar en la parte ya ordenada



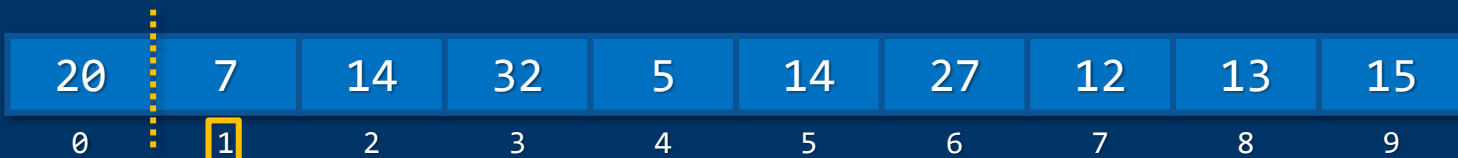
Ordenación por inserción

Ordenación de arrays por inserción

Situación inicial: Lista ordenada con un solo elemento (primero)



*Desde el segundo elemento del array hasta el último:
Localizar el primer elemento mayor en lo ya ordenado*



Primer elemento mayor o igual: índice 0

nuevo 7



Ordenación por inserción

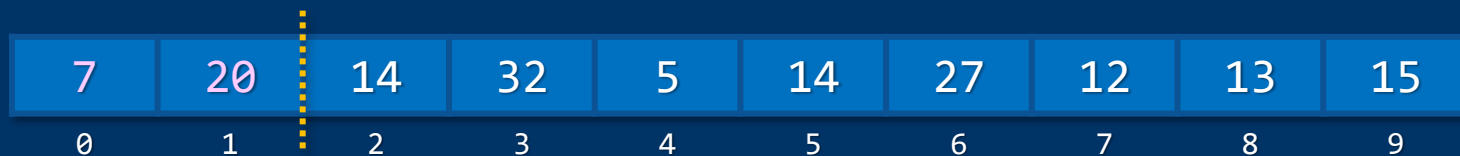
Ordenación de arrays por inserción

...

*Desplazar a la derecha los ordenados desde ese lugar
Insertar el nuevo en la posición que queda libre*



nuevo 7



nuevo 7



Ordenación de arrays por inserción

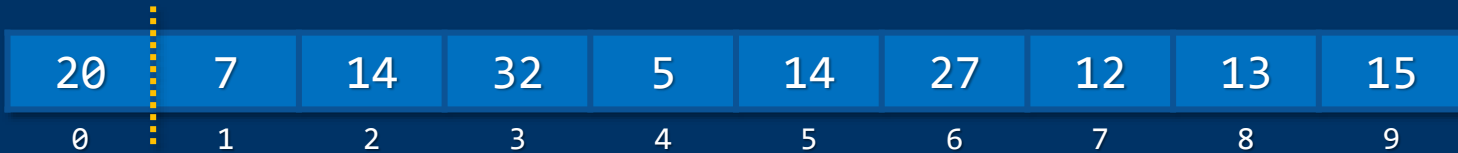
Implementación

```
...
int nuevo, pos;
// Desde el segundo elemento hasta el último...
for (int i = 1; i < N; i++) {
    nuevo = lista[i];
    pos = 0;
    while ((pos < i) && !(lista[pos] > nuevo)) {
        pos++;
    }
    // pos: índice del primer mayor; i si no lo hay
    for (int j = i; j > pos; j--) {
        lista[j] = lista[j - 1];
    }
    lista[pos] = nuevo;
}
```

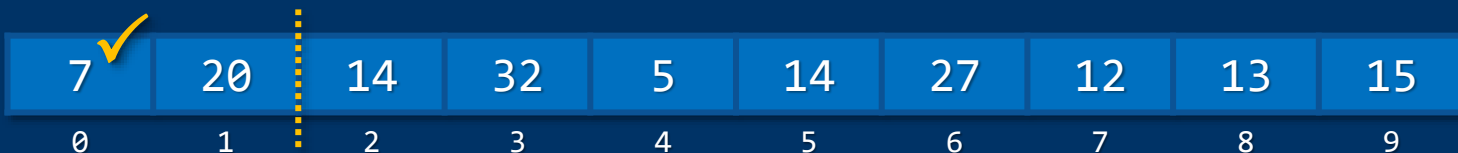
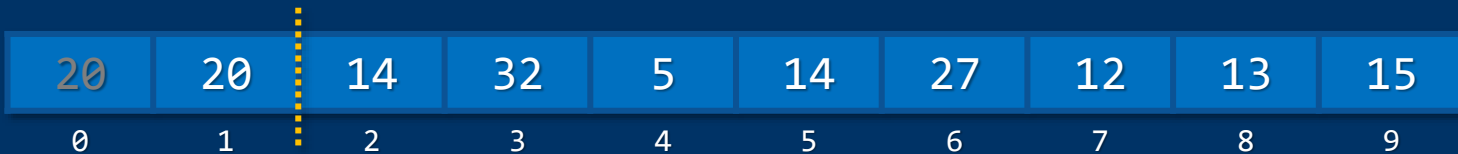
```
const int N = 15;
typedef int tLista[N];
tLista lista;
```



Ordenación de arrays por inserción



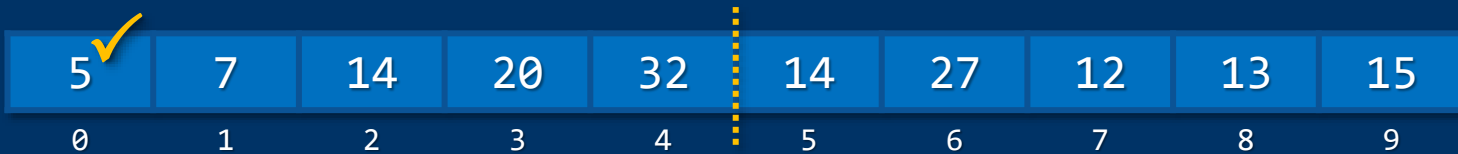
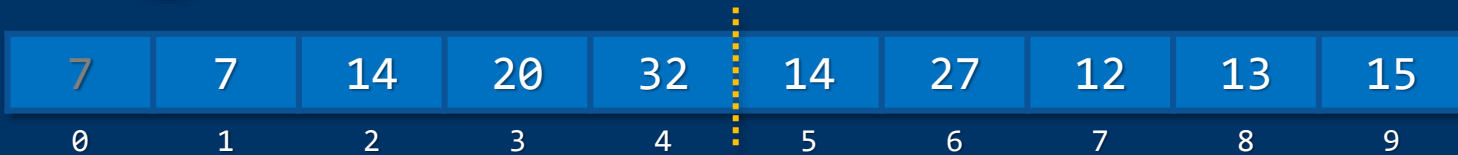
i 1 pos 0 nuevo 7



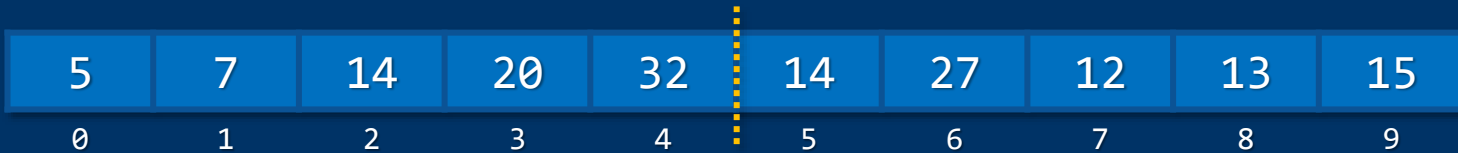
Ordenación de arrays por inserción



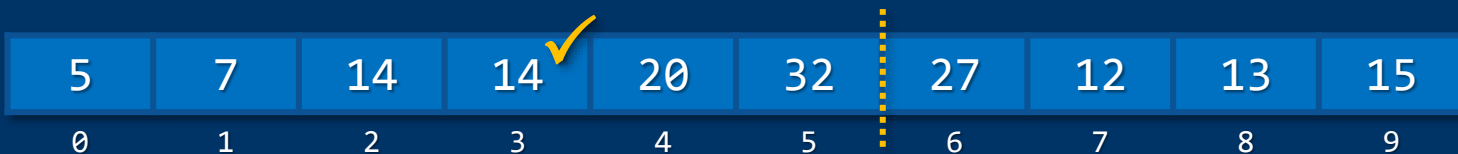
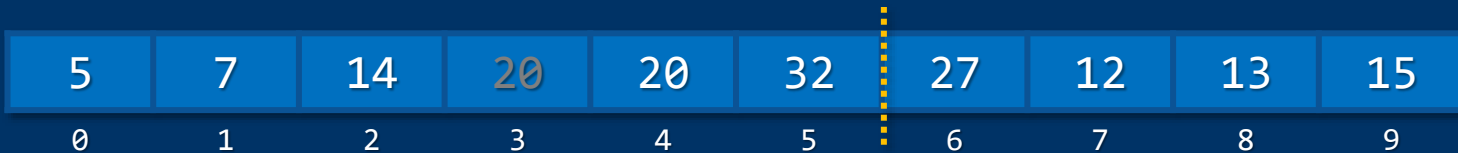
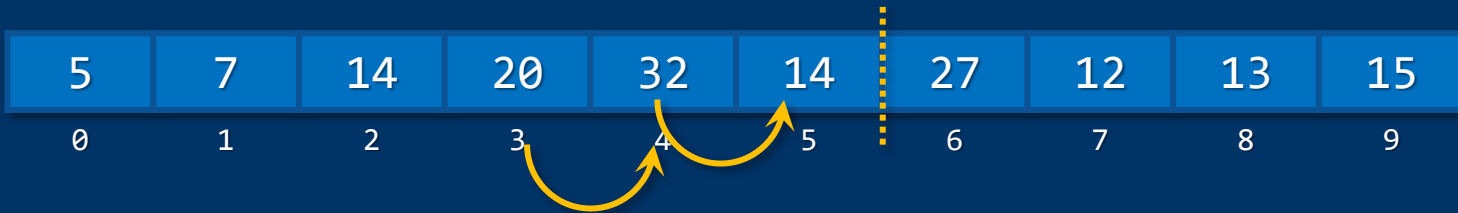
i 4 pos 0 nuevo 5



Ordenación de arrays por inserción



i 5 pos 3 nuevo 14



Fundamentos de la programación

Algoritmo de ordenación por inserción con intercambios



Ordenación por inserción con intercambios

La inserción de cada elemento se puede realizar con comparaciones e intercambios

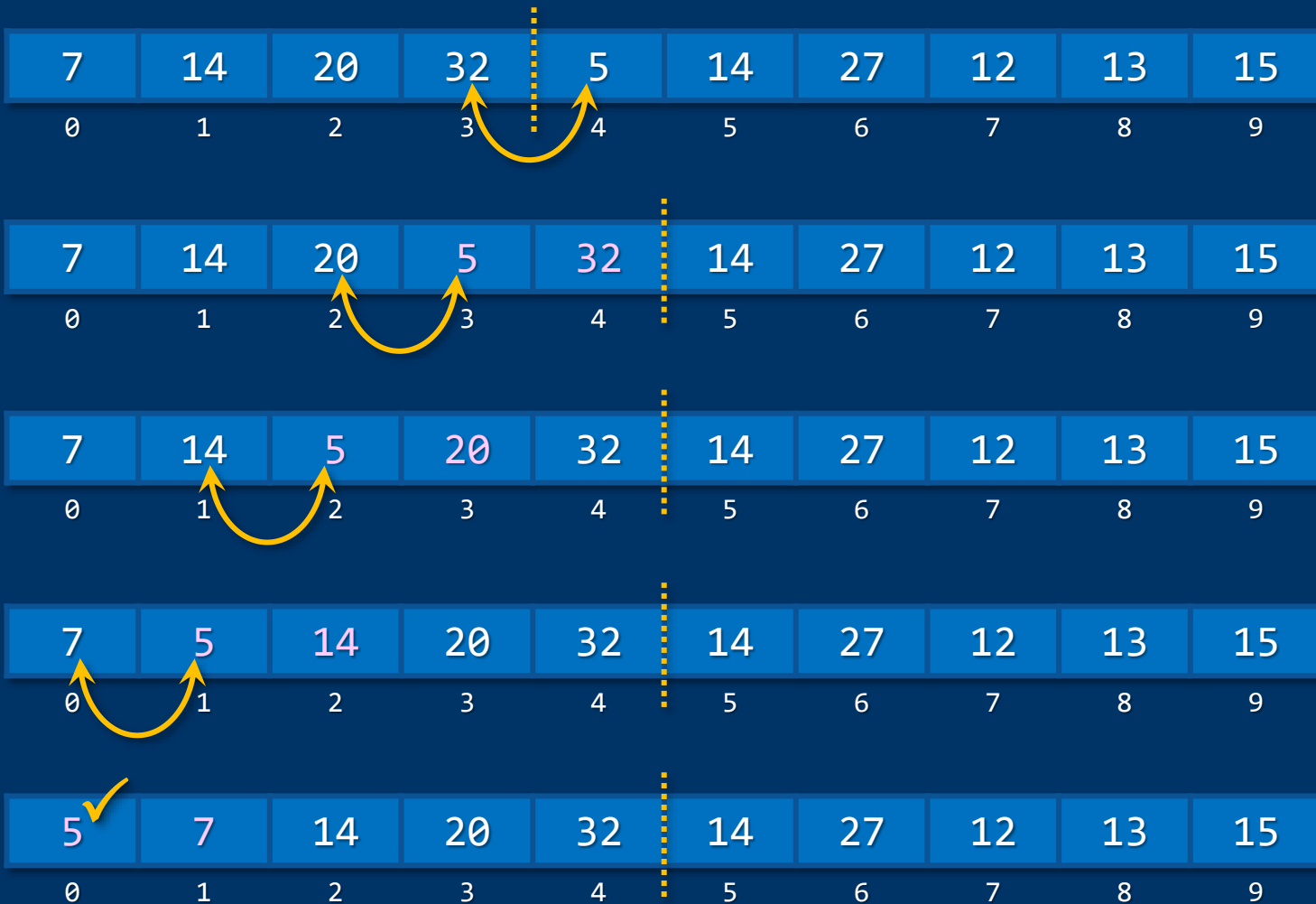
Desde el segundo elemento hasta el último:

Desde la posición del nuevo elemento a insertar:

*Mientras el anterior sea **mayor**, intercambiar*



Ordenación por inserción con intercambios



Ordenación por inserción con intercambios

```
const int N = 15;
typedef int tLista[N];
tLista lista;
```

```
...
int tmp, pos;
// Desde el segundo elemento hasta el último...
for (int i = 1; i < N; i++) {
    pos = i;
    // Mientras no al principio y anterior mayor...
    while ((pos > 0) && (lista[pos - 1] > lista[pos])) {
        // Intercambiar...
        tmp = lista[pos];
        lista[pos] = lista[pos - 1];
        lista[pos - 1] = tmp;
        pos--; // Posición anterior
    }
}
```



Ordenación por inserción con intercambios

insercion.cpp

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

const int N = 100;
typedef int TArray[N];
typedef struct { // Lista de longitud variable
    TArray elementos;
    int contador;
} tLista;

int main() {
    tLista lista;
    ifstream archivo;
    int dato, pos, tmp;
    lista.contador = 0;
    ...
}
```



Ordenación por inserción con intercambios

```
archivo.open("insercion.txt");
if (!archivo.is_open()) {
    cout << "Error de apertura de archivo!" << endl;
}
else {
    archivo >> dato;
    while ((lista.contador < N) && (dato != -1)) {
        // Centinela -1 al final
        lista.elementos[lista.contador] = dato;
        lista.contador++;
        archivo >> dato;
    }
    archivo.close();
    // Si hay más de N ignoramos el resto
    cout << "Antes de ordenar:" << endl;
    for (int i = 0; i < lista.contador; i++) {
        cout << lista.elementos[i] << " ";
    }
    cout << endl;
    ...
}
```



Ordenación por inserción con intercambios

```
for (int i = 1; i < lista.contador; i++) {
    pos = i;
    while ((pos > 0)
        && (lista.elementos[pos-1] > lista.elementos[pos]))
    {
        tmp = lista.elementos[pos];
        lista.elementos[pos] = lista.elementos[pos - 1];
        lista.elementos[pos - 1] = tmp;
        pos--;
    }
}
cout << "Después de ordenar:" << endl;
for (int i = 0; i < lista.contador; i++) {
    cout << lista.elementos[i] << " ";
}
cout << endl;
}
return 0;
}
```



Ordenación por inserción con intercambios

Consideración de implementación

¿Operador relacional adecuado?

`lista[pos - 1] > lista[pos]` ¿> o >=?

Con >= se realizan intercambios inútiles:



¡Intercambio inútil!



Fundamentos de la programación

Claves de ordenación



Ordenación por inserción

Claves de ordenación

Elementos que son estructuras con varios campos:

```
const int N = 15;
typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tDato;
typedef tDato tLista[N];
tLista lista;
```

Clave de ordenación:

Campo en el que se basan las comparaciones



Ordenación por inserción

Claves de ordenación

```
tDato tmp;
while ((pos > 0)
      && (lista[pos - 1].nombre > lista[pos].nombre)) {
    tmp = lista[pos];
    lista[pos] = lista[pos - 1];
    lista[pos - 1] = tmp;
    pos--;
}
```

Comparación: campo concreto

Intercambio: elementos completos




Ordenación por inserción

Claves de ordenación

Función para la comparación:

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

```
tDato tmp;  
while ((pos > 0) && (lista[pos - 1] > lista[pos])) {  
    tmp = lista[pos];  
    lista[pos] = lista[pos - 1];  
    lista[pos - 1] = tmp;  
    pos--;  
}
```



Claves de ordenación

```
#include <iostream>
#include <string>
using namespace std;
#include <fstream>
#include <iomanip>
const int N = 15;
typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tDato;
typedef tDato tArray[N];
typedef struct {
    tArray datos;
    int cont;
} tLista;
...
```



Ordenación por inserción

```
void mostrar(tLista lista);
bool operator>(tDato opIzq, tDato opDer);

int main() {
    tLista lista;
    ifstream archivo;
    lista.cont = 0;
    archivo.open("datos.txt");
    if (!archivo.is_open()) {
        cout << "Error de apertura del archivo!" << endl;
    }
    else {
        tDato dato;
        archivo >> dato.codigo;
        while ((lista.cont < N) && (dato.codigo != -1)) {
            archivo >> dato.nombre >> dato.sueldo;
            lista.datos[lista.cont] = dato;
            lista.cont++;
            archivo >> dato.codigo;
        }
        archivo.close();
        ...
    }
}
```



Ordenación por inserción

```
cout << "Antes de ordenar:" << endl;
mostrar(lista);
for (int i = 1; i < lista.cont; i++) {
// Desde el segundo elemento hasta el último
    int pos = i;
    while ((pos > 0)
            && (lista.datos[pos-1] > lista.datos[pos])) {
        tDato tmp;
        tmp = lista.datos[pos];
        lista.datos[pos] = lista.datos[pos - 1];
        lista.datos[pos - 1] = tmp;
        pos--;
    }
}
cout << "Después de ordenar:" << endl;
mostrar(lista);
}
return 0;
}
```

...



Ordenación por inserción

```
void mostrar(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;  
    }  
}
```

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

Antes de ordenar:

10000	Sergei	100000.00
10000	Hernández	150000.00
11111	Benítez	100000.00
11111	Urpiano	90000.00
11111	Pérez	90000.00
11111	Durán	120000.00
12345	Álvarez	120000.00
12345	Gómez	100000.00
12345	Sánchez	90000.00
12345	Turégano	100000.00
21112	Domínguez	90000.00
21112	Jiménez	100000.00
22222	Fernández	120000.00
33333	Tarazona	120000.00

Después de ordenar:

11111	Benítez	100000.00
21112	Domínguez	90000.00
11111	Durán	120000.00
22222	Fernández	120000.00
12345	Gómez	100000.00
10000	Hernández	150000.00
21112	Jiménez	100000.00
11111	Pérez	90000.00
10000	Sergei	100000.00
12345	Sánchez	90000.00
33333	Tarazona	120000.00
12345	Turégano	100000.00
11111	Urpiano	90000.00
12345	Álvarez	120000.00

Cambia a codigo o sueldo para ordenar por otros campos



Fundamentos de la programación

Estabilidad de la ordenación



Estabilidad de la ordenación

Algoritmos de ordenación estables

Al ordenar por otra clave una lista ya ordenada, la segunda ordenación preserva el orden de la primera

tDato: tres posibles claves de ordenación (campos)

Código

Nombre

Sueldo

Lista ordenada por Nombre →

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



Estabilidad de la ordenación

Ordenamos ahora por el campo Código:

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

10000	Hernández	150000
10000	Sergei	100000
11111	Benítez	100000
11111	Durán	120000
11111	Pérez	90000
11111	Urpiano	90000
12345	Álvarez	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

Estable:

Los nombres mantienen sus posiciones relativas



Estabilidad de la ordenación

Ordenación por inserción

Estable siempre que utilicemos $< o >$ Con $\leq o \geq$ no es estable

Ordenamos por sueldo:

A igual sueldo, ordenado por códigos y a igual código, por nombres

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



Fundamentos de la programación

Complejidad y eficiencia



Complejidad y eficiencia

Casos de estudio para los algoritmos de ordenación

- ✓ Lista inicialmente ordenada

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

- ✓ Lista inicialmente ordenada al revés

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

- ✓ Lista con disposición inicial aleatoria

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

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



Complejidad y eficiencia

Ordenaciones naturales

Si el algoritmo 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 que busca el lugar primero y luego desplaza:
No hay desplazamientos; mismo número de comparaciones
Comportamiento no natural
- ✓ Versión con intercambios:
Trabaja mucho menos; basta una comparación cada vez
Comportamiento natural



Complejidad y eficiencia

Elección de un algoritmo de ordenación

¿Cómo de bueno es cada algoritmo?

¿Cuánto tarda en comparación con otros algoritmos?

Algoritmos más eficientes: los de menor complejidad

Tardan menos en realizar la misma tarea

Comparamos en orden de complejidad: $O()$

En función de la dimensión de la lista a ordenar: N

$O() = f(N)$

Operaciones que realiza el algoritmo de ordenación:

- ✓ Comparaciones
- ✓ Intercambios

Asumimos que tardan un tiempo similar



Complejidad y eficiencia

Cálculo de la complejidad

Ordenación por inserción (con intercambios):

```
...
for (int i = 1; i < N; i++) {
    int pos = i;
    while ((pos > 0) && (lista[pos - 1] > lista[pos])) {
        int tmp;
        tmp = lista[pos];
        lista[pos] = lista[pos - 1];
        lista[pos - 1] = tmp;
        pos--;
    }
}
```

Comparación

Intercambio

Intercambios y comparaciones:

Tantos como ciclos realicen los correspondientes bucles



Complejidad y eficiencia

Cálculo de la complejidad

```
...  
                                N - 1 ciclos  
for (int i = 1; i < N; i++) {  
    int pos = i;                                N° variable de ciclos  
    while ((pos > 0) && (lista[pos - 1] > lista[pos])) {  
        int tmp;  
        tmp = lista[pos];  
        lista[pos] = lista[pos - 1];  
        lista[pos - 1] = tmp;  
        pos--;  
    }  
}
```

Caso en el que el `while` se ejecuta más: *caso peor*

Caso en el que se ejecuta menos: *caso mejor*



Complejidad y eficiencia

Cálculo de la complejidad

- ✓ Caso mejor: lista inicialmente ordenada
La primera comparación falla: ningún intercambio
 $(N - 1) * (1 \text{ comparación} + 0 \text{ intercambios}) = N - 1 \rightarrow O(N)$
- ✓ Caso peor: lista inicialmente ordenada al revés
Para cada pos, entre i y 1: 1 comparación y 1 intercambio
 $1 + 2 + 3 + 4 + \dots + (N - 1)$
 $((N - 1) + 1) * (N - 1) / 2$
 $N * (N - 1) / 2$
 $(N^2 - N) / 2 \rightarrow O(N^2)$

Notación *O grande*: orden de complejidad en base a N
El término en N que más rápidamente crece al crecer N
En el caso peor, N^2 crece más rápido que N $\rightarrow O(N^2)$
(Ignoramos las constantes, como 2)



Complejidad y eficiencia

Ordenación por inserción (con intercambios)

- ✓ Caso mejor: $O(N)$
- ✓ Caso peor: $O(N^2)$

Caso medio (distribución aleatoria de los elementos): $O(N^2)$

Hay algoritmos de ordenación mejores



Complejidad y eficiencia

Órdenes de complejidad

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

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
...		



Fundamentos de la programación

Ordenación por selección directa



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan



Lista ordenada:



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan

Lista desordenada: 

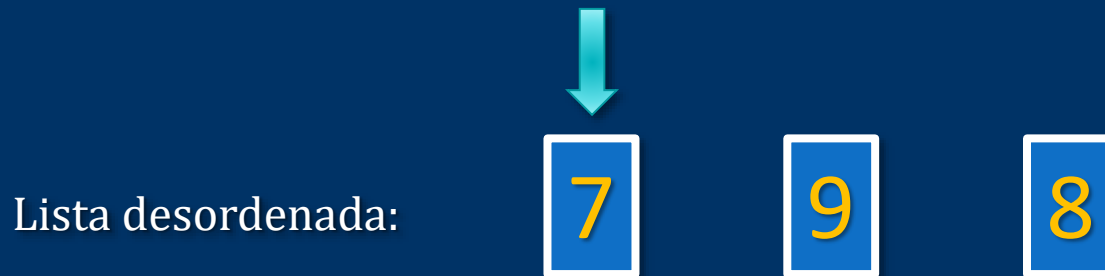
Lista ordenada: 



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan

Lista desordenada:



Lista ordenada:



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan

Lista desordenada:



Lista ordenada:



Ordenación por selección directa

Algoritmo de ordenación por selección directa

Seleccionar el siguiente elemento menor de los que quedan

Lista desordenada:

!!! LISTA ORDENADA !!!

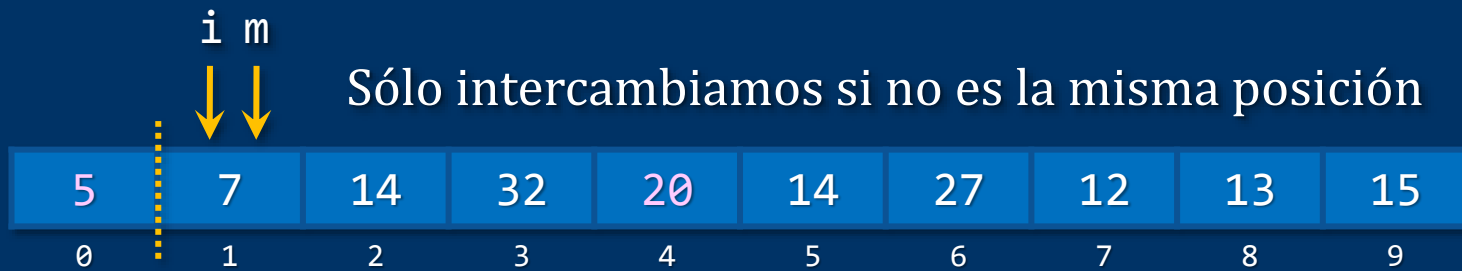
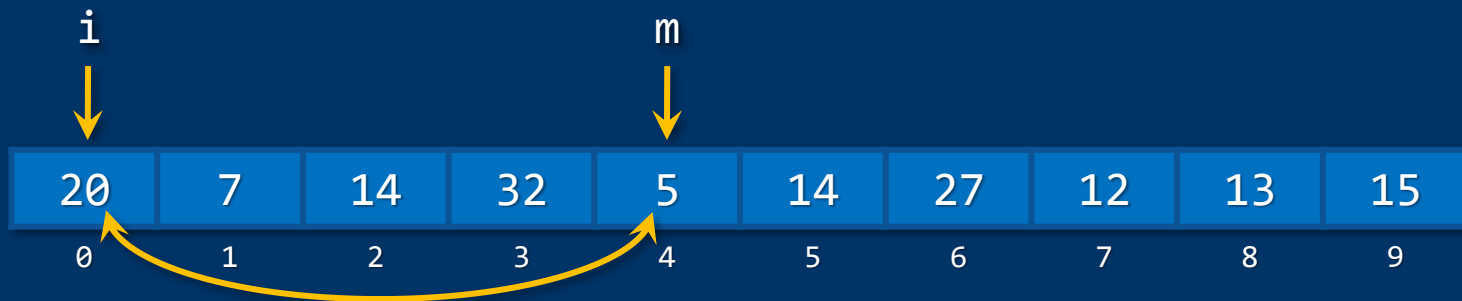
Lista ordenada:



Ordenación por selección directa

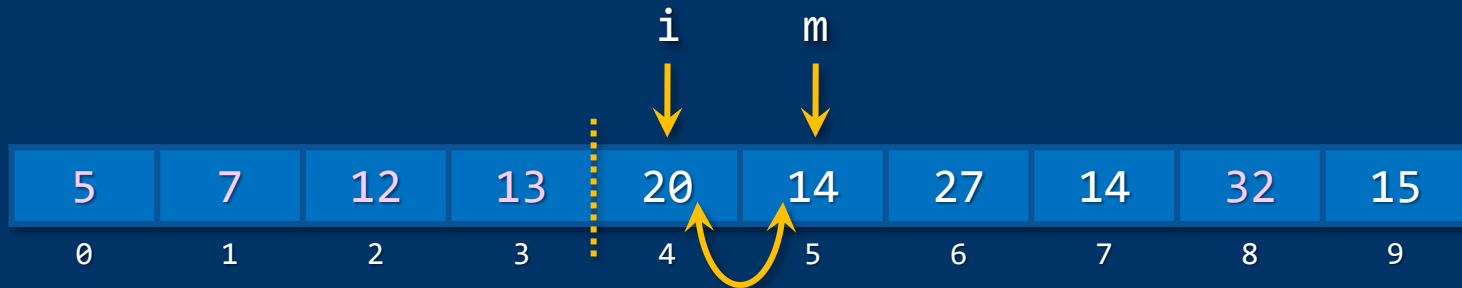
Ordenación de un array por selección directa

Desde el primer elemento ($i = 0$) hasta el penúltimo ($N-2$):
Menor elemento (en m) entre $i + 1$ y el último ($N-1$)
Intercambiar los elementos en i y m si no son el mismo



Ordenación por selección directa

Ordenación de un array por selección directa



Implementación

```
const int N = 15;
typedef int tLista[N];
tLista lista;
```

```
// Desde el primer elemento hasta el penúltimo...
for (int i = 0; i < N - 1; i++) {
    int menor = i;
    // Desde i + 1 hasta el final...
    for (int j = i + 1; j < N; j++) {
        if (lista[j] < lista[menor]) {
            menor = j;
        }
    }
    if (menor > i) {
        int tmp;
        tmp = lista[i];
        lista[i] = lista[menor];
        lista[menor] = tmp;
    }
}
```



Ordenación por selección directa

Complejidad de la ordenación por selección directa

¿Cuántas comparaciones se realizan?

Bucle externo: $N - 1$ ciclos

Tantas comparaciones como elementos queden en la lista:

$$(N - 1) + (N - 2) + (N - 3) + \dots + 3 + 2 + 1 =$$

$$N \times (N - 1) / 2 = (N^2 - N) / 2 \rightarrow O(N^2)$$

Mismo número de comparaciones en todos los casos

Complejidad: $O(N^2)$ Igual que el método de inserción

Algo mejor (menos intercambios; uno en cada paso)

No es estable: intercambios “a larga distancia”

No se garantiza que se mantenga el mismo orden relativo original

Comportamiento no natural (trabaja siempre lo mismo)



Fundamentos de la programación

Método de la burbuja



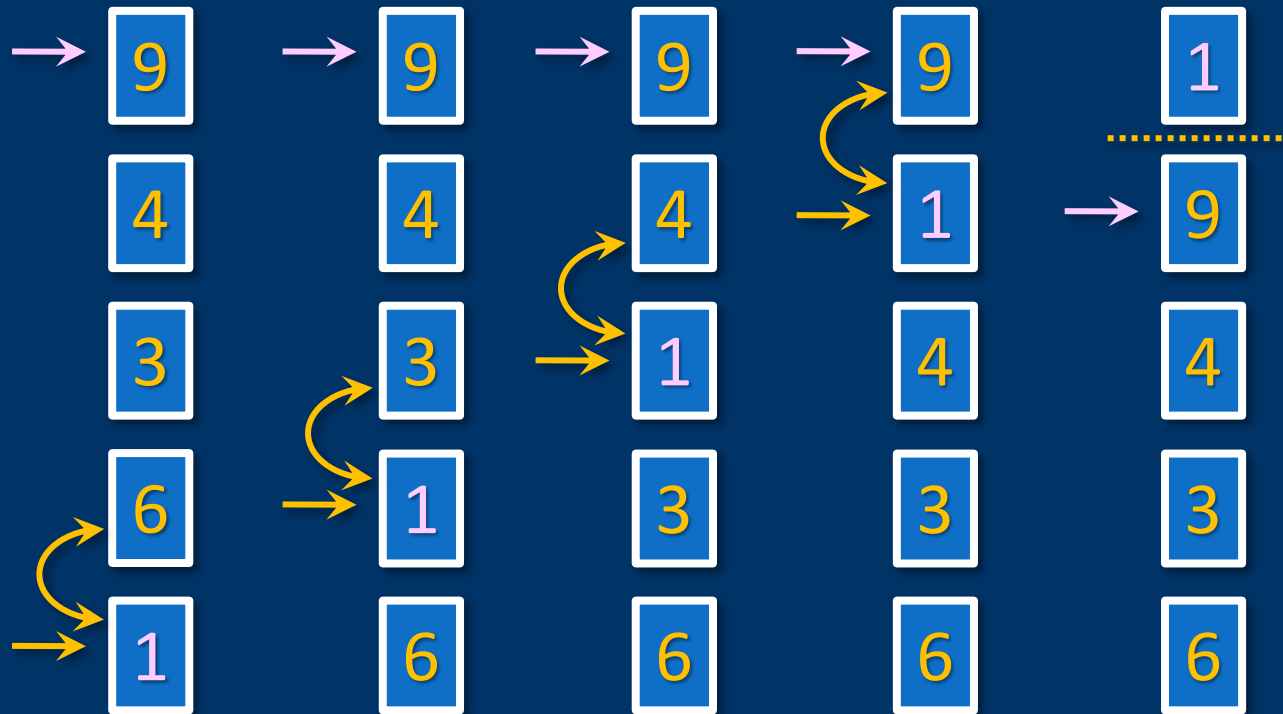
Método de la burbuja



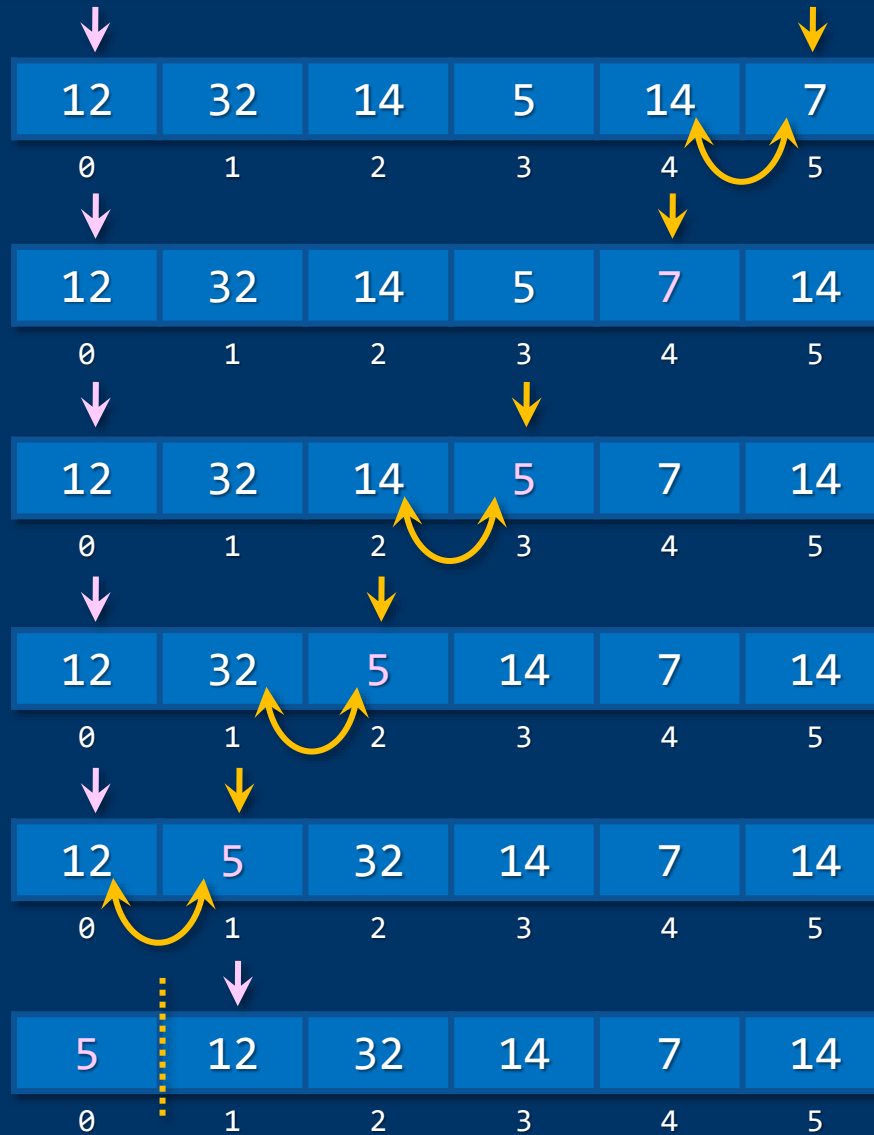
Algoritmo de ordenación por el método de la burbuja

Variación del método de selección directa

El elemento menor va *ascendiendo* hasta alcanzar su posición



Método de la burbuja



Ordenación de un array por el método de la burbuja

Desde el primero ($i = 0$), hasta el penúltimo ($N - 2$):

Desde el último ($j = N - 1$), hasta $i + 1$:

Si elemento en $j <$ elemento en $j - 1$, intercambiarlos

```
...
int tmp;
// Del primero al penúltimo...
for (int i = 0; i < N - 1; i++) {
    // Desde el último hasta el siguiente a i...
    for (int j = N - 1; j > i; j--) {
        if (lista[j] < lista[j - 1]) {
            tmp = lista[j];
            lista[j] = lista[j - 1];
            lista[j - 1] = tmp;
        }
    }
}
```

```
const int N = 10;
typedef int tLista[N];
tLista lista;
```



Método de la burbuja

Algoritmo de ordenación por el método de la burbuja

Complejidad: $O(N^2)$

Comportamiento no natural

Estable (mantiene el orden relativo)

Mejora:

Si en un paso del bucle exterior no ha habido intercambios:

La lista ya está ordenada (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



Método de la burbuja mejorado

burbuja2.cpp

```
bool inter = true;
int i = 0;
// Desde el 1º hasta el penúltimo si hay intercambios...
while ((i < N - 1) && inter) {
    inter = false;
    // Desde el último hasta el siguiente a i...
    for (int j = N - 1; j > i; j--) {
        if (lista[j] < lista[j - 1]) {
            int tmp;
            tmp = lista[j];
            lista[j] = lista[j - 1];
            lista[j - 1] = tmp;
            inter = true;
        }
    }
    if (inter) {
        i++;
    }
}
```

Esta variación sí tiene un comportamiento natural



Fundamentos de la programación

Listas ordenadas



Listas ordenadas

Gestión de listas ordenadas

Casi todas las tareas se realizan igual que en listas sin orden

Operaciones que tengan en cuenta el orden:

- ✓ Inserción de un nuevo elemento: debe seguir en orden
- ✓ Búsquedas más eficientes

¿Y la carga desde archivo?

- ✓ Si los elementos se guardaron en orden: se lee igual
- ✓ Si los elementos no están ordenados en el archivo: insertar



Declaraciones: Iguales que para listas sin orden

```
const int N = 20;
```

```
typedef struct {  
    int codigo;  
    string nombre;  
    double sueldo;  
} tRegistro;
```

```
typedef tRegistro tArray[N];
```

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



Gestión de listas ordenadas

Subprogramas: Misma declaración que para listas sin orden

```
void mostrarDato(int pos, tRegistro registro);
void mostrar(tLista lista);
bool operator>(tRegistro opIzq, tRegistro opDer);
bool operator<(tRegistro opIzq, tRegistro opDer);
tRegistro nuevo();
void insertar(tLista &lista, tRegistro registro, bool &ok);
void eliminar(tLista &lista, int pos, bool &ok); // pos = 1..N
int buscar(tLista lista, string nombre);
void cargar(tLista &lista, bool &ok);
void guardar(tLista lista);
```



Gestión de listas ordenadas

Nuevas implementaciones:

- ✓ Operadores relacionales
- ✓ Inserción (mantener el orden)
- ✓ Búsqueda (más eficiente)

Se guarda la lista en orden, por lo que cargar() no cambia

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

```
bool operator<(tRegistro opIzq, tRegistro opDer) {  
    return opIzq.nombre < opDer.nombre;  
}
```



Gestión de listas ordenadas

```
void insertar(tLista &lista, tRegistro registro, bool &ok) {
    ok = true;
    if (lista.cont == N) {
        ok = false; // lista llena
    }
    else {
        int i = 0;
        while ((i < lista.cont) && (lista.registros[i] < registro)) {
            i++;
        }
        // Insertamos en la posición i (primer mayor o igual)
        for (int j = lista.cont; j > i; j--) {
            // Desplazamos una posición a la derecha
            lista.registros[j] = lista.registros[j - 1];
        }
        lista.registros[i] = registro;
        lista.cont++;
    }
}
```



Fundamentos de la programación

Búsquedas en listas ordenadas



Búsquedas en listas ordenadas

Búsqueda de un elemento en una secuencia

No ordenada: recorreremos hasta encontrarlo **o al final**

Ordenada: recorreremos hasta encontrarlo **o mayor / al final**

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

Buscamos el 36: al llegar al final sabemos que no está

Buscamos el 17: al llegar al 20 ya sabemos que no está

Condiciones de terminación:

- ✓ Se llega al final
- ✓ Se encuentra el elemento buscado
- ✓ Se encuentra uno mayor
- Mientras no al final y el valor sea menor que el buscado



Búsquedas en listas ordenadas

```
int buscado;
cout << "Valor a buscar: ";
cin >> buscado;
int i = 0;
while ((i < N) && (lista[i] < buscado)) {
    i++;
}
// Ahora, o estamos al final o lista[i] >= buscado
if (i == N) { // Al final: no se ha encontrado
    cout << "No encontrado!" << endl;
}
else if (lista[i] == buscado) { // Encontrado!
    cout << "Encontrado en posición " << i + 1 << endl;
}
else { // Hemos encontrado uno mayor
    cout << "No encontrado!" << endl;
}
```

```
const int N = 10;
typedef int tLista[N];
tLista lista;
```

Complejidad: $O(N)$



Fundamentos de la programación

Búsqueda binaria



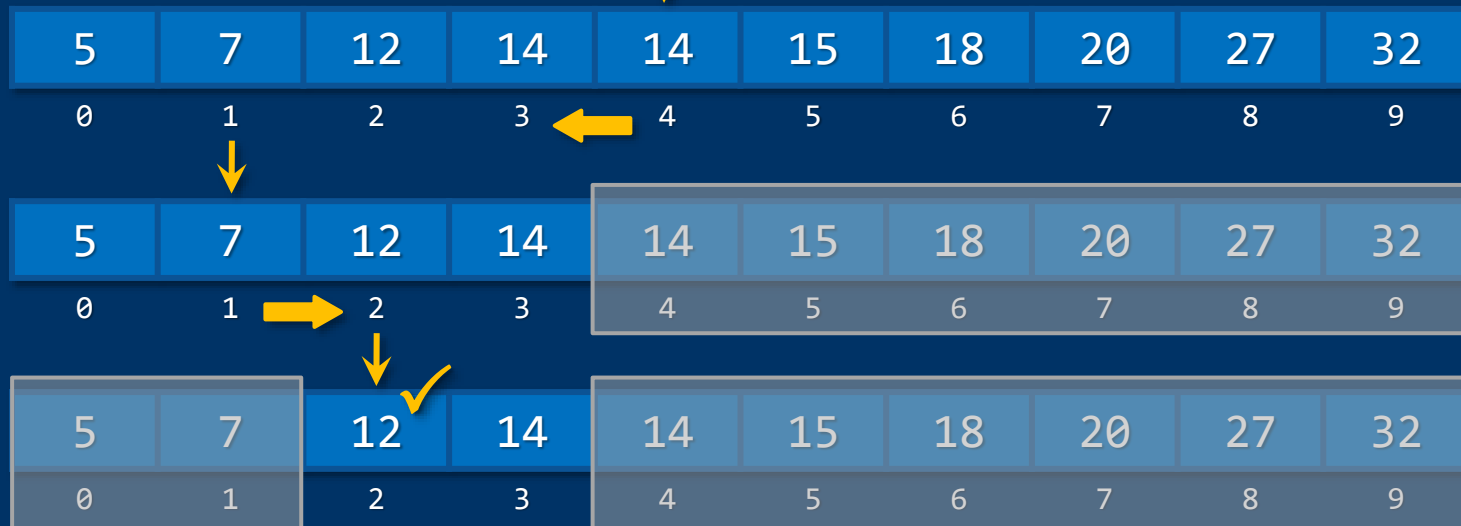
Búsqueda binaria

Búsqueda mucho más rápida que aprovecha la ordenación

*Comparar con el valor que esté en el medio de la lista:
Si es el que se busca, terminar
Si no, si es mayor, buscar en la primera mitad de la lista
Si no, si es menor, buscar en la segunda mitad de la lista
Repetir hasta encontrarlo o no quede sublista donde buscar*

Buscamos el 12

↓ Elemento mitad



Búsqueda binaria

Vamos buscando en sublistas cada vez más pequeñas (mitades)

Delimitamos el segmento de la lista donde buscar

Inicialmente tenemos toda la lista:



Índice del elemento en la mitad: $\text{mitad} = (\text{ini} + \text{fin}) / 2$

Si no se encuentra, ¿dónde seguir buscando?

Buscado < elemento en la mitad: $\text{fin} = \text{mitad} - 1$

Buscado > elemento en la mitad: $\text{ini} = \text{mitad} + 1$

Si $\text{ini} > \text{fin}$, no queda dónde buscar

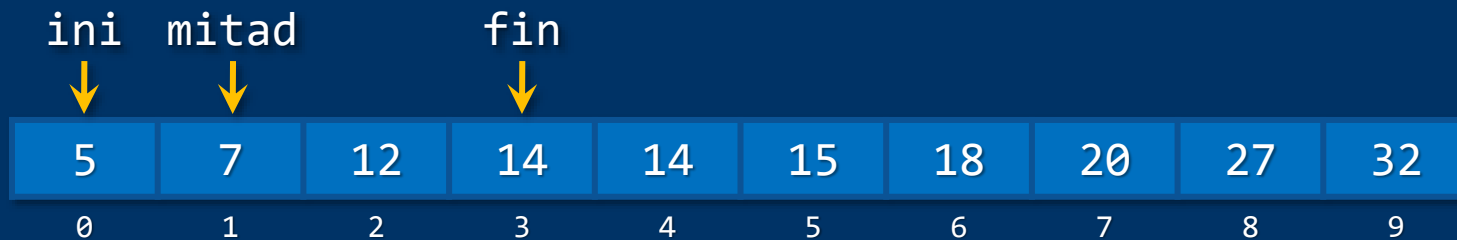


Búsqueda binaria

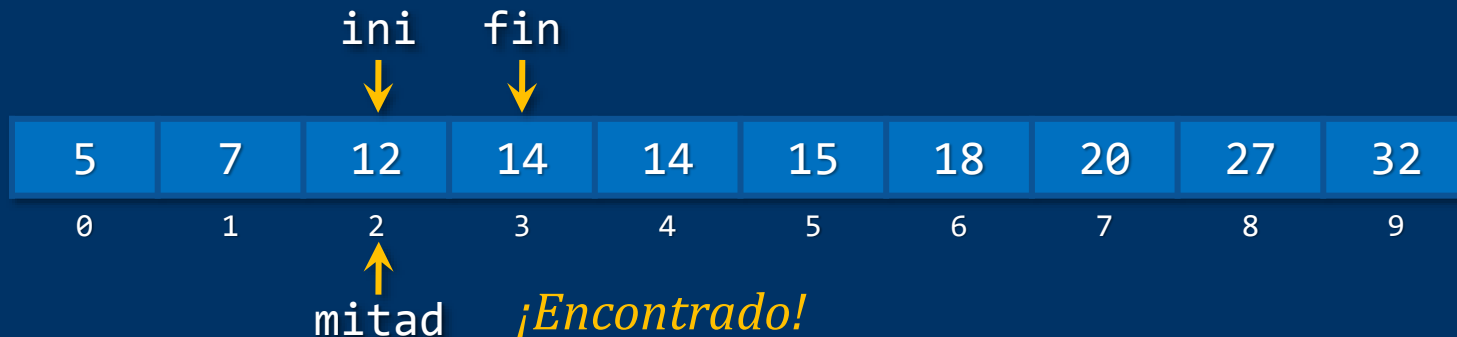
Buscamos el 12



$12 < \text{lista}[\text{mitad}] \rightarrow \text{fin} = \text{mitad} - 1$



$12 > \text{lista}[\text{mitad}] \rightarrow \text{ini} = \text{mitad} + 1$



Búsqueda binaria

Si el elemento no está, nos quedamos sin sublista: $ini > fin$

Para el 13: mitad
ini fin

5	7	12	14	14	15	18	20	27	32
0	1	2	3	4	5	6	7	8	9

$13 > lista[mitad] \rightarrow ini = mitad + 1$

mitad
ini
fin

5	7	12	14	14	15	18	20	27	32
0	1	2	3	4	5	6	7	8	9

$13 < lista[mitad] \rightarrow fin = mitad - 1 \rightarrow 2$

!!! $ini > fin$!!! No hay dónde seguir buscando \rightarrow No está



Búsqueda binaria

Implementación

```
int 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;
    }
} // Si se ha encontrado, está en [mitad]
```

```
const int N = 10;
typedef int tLista[N];
tLista lista;
```



Búsqueda binaria

binaria.cpp

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

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

int buscar(tLista lista, int buscado);

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



Búsqueda binaria

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



Búsqueda binaria

```
int buscar(tLista lista, int buscado) {
    int pos = -1, 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 pos;
}
```



Búsqueda binaria

Complejidad

¿Qué orden de complejidad tiene la búsqueda binaria?

Caso peor:

No está o se encuentra en una sublista de 1 elemento

Nº de comparaciones = Nº de mitades que podemos hacer

$N / 2, N / 4, N / 8, N / 16, \dots, 8, 4, 2, 1$

$\equiv 1, 2, 4, 8, \dots, N / 16, N / 8, N / 4, N / 2$

Si hacemos que N sea igual a 2^k :

$2^0, 2^1, 2^2, 2^3, \dots, 2^{k-4}, 2^{k-3}, 2^{k-2}, 2^{k-1}$

Nº de elementos de esa serie: k

Nº de comparaciones = k $N = 2^k \rightarrow k = \log_2 N$

Complejidad: $O(\log_2 N)$ Mucho más rápida que $O(N)$






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.

