

Fundamentos de la programación

Algoritmos de ordenación y gestión de datos ordenados

Facultad de Informática
Universidad Complutense

Ana Gil Luezas, y modificadas por Sonia Estévez
(Adaptadas del original de Luis Hernández Yáñez)



Índice

| | |
|------------------------------------|----|
| Algoritmos de ordenación de arrays | 2 |
| Ordenación por inserción | 4 |
| Claves de ordenación | 11 |
| Estabilidad de la ordenación | 18 |
| Complejidad y eficiencia | 21 |
| Ordenación por selección | 28 |
| Método de la burbuja | 33 |
| Operaciones con arrays ordenados | |
| Búsqueda secuencial | 40 |
| Búsqueda binaria | 42 |
| Inserción y eliminación | 51 |
| Mezcla | 62 |



Algoritmos de ordenación

Ordenación de N datos en un array

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
(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 |

`array[i] <= array[i + 1]`

Tarea muy habitual: para mostrar los datos en orden, facilitar las búsquedas, ...
Diferentes formas de hacerlo (algoritmos), con mayor o menor rapidez.



Algoritmos de ordenación

□ *Algoritmos de ordenación básicos:*

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

Los algoritmos se basan en comparaciones y asignaciones.

Hay otros algoritmos de ordenación mejores (Quick sort, Merge sort, ...)



Ordenación por inserción

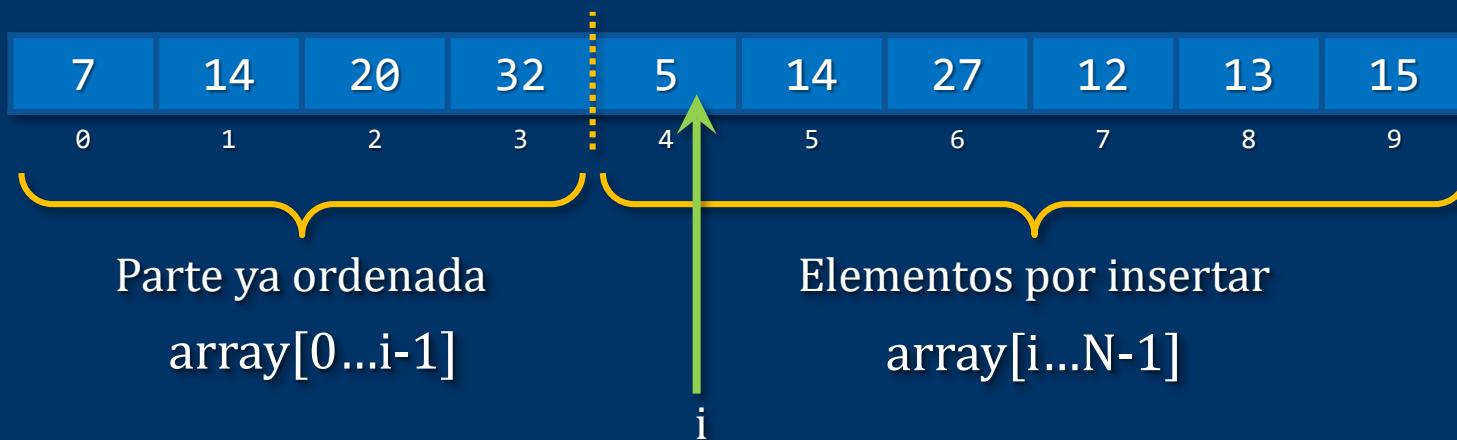


Ordenación por inserción

El array contiene inicialmente los datos desordenados:

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

A medida que se insertan los elementos en su lugar:



Siguiente elemento a insertar en la parte ya ordenada ($i==4$)



Ordenación por inserción

Situación inicial: Parte ordenada con un solo elemento, el primero del array (array[0...(1-1)]):

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

Desde el segundo elemento hasta el último:

Obtenemos el elemento a insertar (array[i]) y

Lo insertamos ordenadamente en la parte ordenada.

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



elemento 7

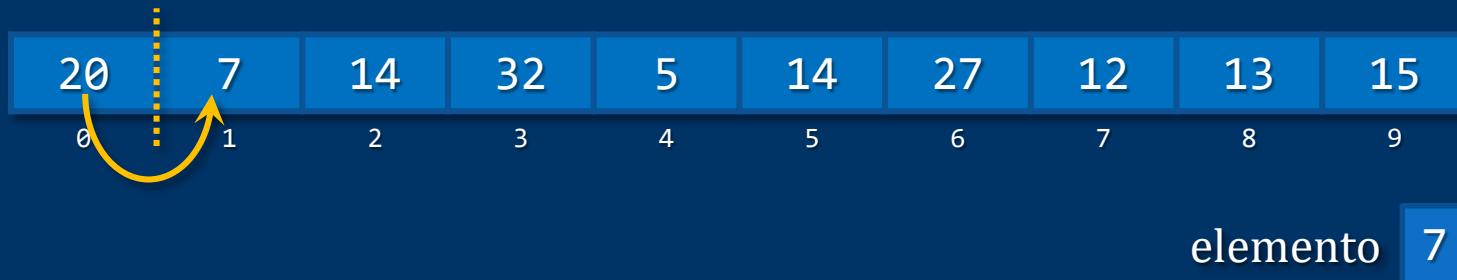
Último elemento de la parte ordenada (el mayor): índice 0



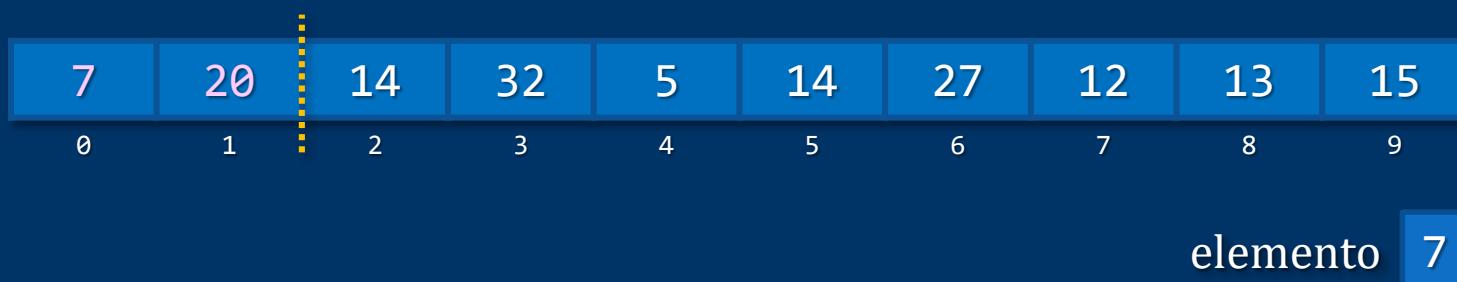
Ordenación por inserción

Insertar ordenadamente el elemento:

Desplazamos una posición a la derecha todos los elementos, de la parte ya ordenada, que sean mayores que elemento.



Y colocamos el nuevo elemento en la posición que queda libre.



Ordenación por inserción

```
// Tipo de datos de los elementos (con operadores: < y =)
typedef ... tDato ... ;
void ordenarInsercion(tDato tabla[], int cont) {
//Parte ordenada tabla[0...i-1], parte por procesar tabla[i...N-1]
    tDato elemento; int j;
    for (int i= 1; i< cont; ++i){// Desde el segundo hasta el último
        elemento = tabla[i]; // Elemento a insertar
        j = i; // Desplazar los mayores de la parte ordenada
        while ((j > 0) && (elemento < tabla[j-1])){
            tabla[j] = tabla[j-1];
            --j;
        }
        tabla[j] = elemento; // Colocar en el hueco
    } // parte ordenada tabla[0 ... N-1]
}
```



N es el valor de cont



Ordenación por inserción

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

i **1** elemento **7**

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

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

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



Ordenación por inserción

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

i 4 elemento 5

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



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

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



Ordenación por inserción

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

i **5** elemento **14**

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



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

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



Claves de ordenación



Claves de ordenación

Cuando los elementos de las listas que hay que ordenar son estructuras.

```
const int T = N;
typedef struct { //con operadores: < y =
    int codigo;
    string nombre;
    double sueldo;
} tDato;
// typedef tDato tTabla[T];
```

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

```
tDato elemento = tabla[i]; j = i;
while ((j > 0) && (elemento.nombre < tabla[j-1].nombre)) {
    tabla[j] = tabla[j-1];
    --j;
}
tabla[j] = elemento; // Colocar en el hueco (if(j!=i))
```



Claves de ordenación

Podemos sobrecargar el operador <:

```
bool operator<(tDato const& datoI, tDato const& datoD){  
    return datoI.nombre < datoD.nombre;  
}
```

```
while ((j > 0) && (elemento < tabla[j - 1])) {  
    tabla[j] = tabla[j-1];  
    --j;  
}
```



Ejemplo

```
// lista.h

const int TM = ...;

typedef struct { // En dato.h
    int codigo;
    string nombre;
    double sueldo;
} tDato;

// typedef tDato tTabla[TM + 1]; // Posible centinela

typedef struct {
    tDato datos[TM + 1];
    int cont;
} tLista;
```



Ejemplo

```
void ordenarInsercion(tDato tabla[], int cont);

inline void ordenarInsercion(tLista & lista){
    ordenarInsercion(lista.datos, lista.cont);
}

inline bool operator<(tDato const& datI,
                      tDato const& datD){ // En dato.h
    return datI.nombre < datD.nombre;
}

void mostrar(tLista const& lista);
void guardar(ofstream & ent, tLista const& lista);
void cargar(ifstream & ent, tLista & lista);

void mostrar(tDato dato); // En dato.h
void guardar(ofstream & ent, tDato const& dato);
void cargar(ifstream & ent, tDato & dato);
```



Ejemplo

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

        ordenarInsercion(lista);

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



Ejemplo

```
// lista.cpp

void ordenarInsercion(tDato tabla[], int cont){
    int i, j; tDato elemento;
    for (i = 1; i < cont; ++i) {
        j = i; elemento = tabla[i];
        while ((j > 0) && (elemento < tabla[j-1])) {
            tabla[j] = tabla[j-1];
            --j;
        }
        tabla[j] = elemento; // if(j!=i)
    }
}
```



Ejemplo

```
bool cargar(ifstream & ent, tDato & dato){ // En dato.cpp
    ent >> dato.codigo >> dato.nombre >> dato.sueldo;
    return !ent.fail() && dato!= Centinela;
}

bool cargar(ifstream & ent, tLista & lista){
    tDato dato; lista.cont = 0; // !!
    bool fin = false;
    while ((lista.cont < TM+1) && !fin) { // Possible centinela
        if (cargar(ent, lista.datos[lista.cont])) ++lista.cont;
        else fin = true;
    }
    if (list.cont == TM+1){// ERROR: Más datos de los que caben
        --lista.cont; // list.datos[list.cont+1] = Centinela;
        return false; }
    else return true;
}
```



Complejidad y eficiencia

Complejidad y eficiencia

Casos de estudio para los algoritmos de ordenación

- ✓ Lista inicialmente ordenada.
- ✓ Lista inicialmente ordenada al revés.
- ✓ Lista con disposición inicial aleatoria



Complejidad y eficiencia

- ✓ Mide la cantidad de recursos en tiempo (y espacio) que necesita un algoritmo para resolver un problema.
- ✓ Serán más eficientes los algoritmos de menor complejidad, los que tarden menos en realizar la misma tarea.
- ✓ Comparamos en orden de complejidad: $O()$
La complejidad algorítmica no proporciona medidas absolutas sino medidas relativas al tamaño del problema.

En base al número de elementos a ordenar : N
Contaremos las operaciones que realiza el algoritmo.

- ✓ Comparaciones
- ✓ Asignaciones

Asumimos que se tarda un tiempo similar .



Complejidad y eficiencia

Ordenación por inserción

Calculo de la complejidad:

```
for (i = 1; i < N; ++i) {
    j = i; elemento = lista[i];
    while ((j > 0) && (elemento < tabla[j-1])) {
        tabla[j] = tabla[j-1];
        --j;
    }
    tabla[j] = elemento; // if (j!=i)
}
```

Comparaciones

Asignaciones

Se realizan tantas asignaciones y comparaciones como ciclos realicen los bucles en los que se encuentran.



Complejidad y eficiencia

Ordenación por inserción

Calculo de la complejidad:

$N - 1$ ciclos

```
for (i = 1; i < N; ++i) {  
    j = i; elemento = lista[i];      Nº variable de ciclos  
    while ((j > 0) && (elemento < tabla[j-1])) {  
  
        tabla[j] = tabla[j-1];  
        --j;  
    }  
    tabla[j] = elemento;  
}
```

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

Caso en que se ejecuta menos: *caso mejor*



Complejidad y eficiencia

Ordenación por inserción

Calculo de la complejidad:

- ✓ Caso mejor: lista inicialmente ordenada. El `while` no se ejecuta ninguna vez, la primera vez que se comprueba la condición falla.

$$(N-1) \cdot (4 \text{ asignaciones} + 3 \text{ comparaciones}) = 7N - 7$$

Ignoramos las constantes, como 7, polinomio de grado 1:

Complejidad lineal $\rightarrow O(N)$

Notación *O grande*: orden de complejidad en base a N.

- ✓ Caso peor: lista inicialmente ordenada al revés.

El `while` se ejecuta i veces, para cada j entre i y 1. Ignorando las constantes:

$$\begin{aligned} 1 + 2 + 3 + 4 + \dots + (N-1) &= \\ (N-1) \cdot (1 + (N-1)) / 2 &= (N-1) \cdot N / 2 = \\ (N^2 - N) / 2 &\rightarrow O(N^2) \end{aligned}$$

Polinomio de grado 2: Complejidad cuadrática $\rightarrow O(N^2)$



Complejidad y eficiencia

Ordenación por inserción

- ✓ Caso mejor: $O(N)$
- ✓ Caso peor: $O(N^2)$
- ✓ Caso medio (distribución aleatoria de los elementos): $O(N^2)$

Hay algoritmos de ordenación mejores: $O(N \log N)$.

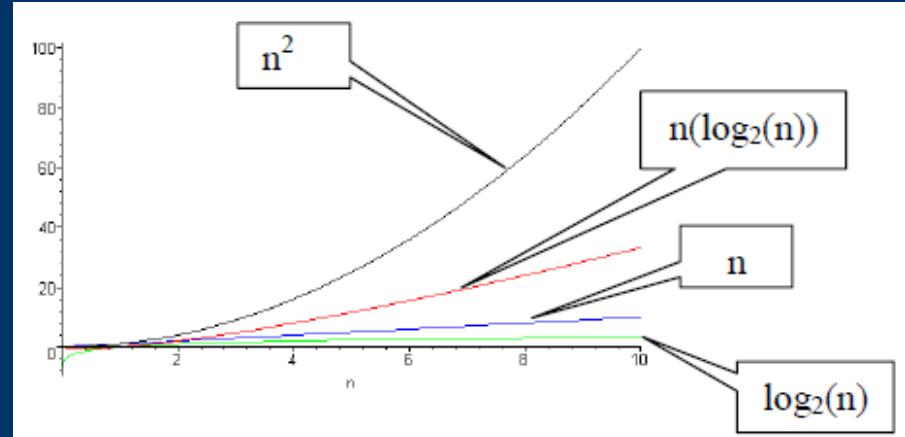


Complejidad y eficiencia

Órdenes de complejidad

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

| N | $\log_2 N$ | $N \cdot \log_2 N$ | N^2 |
|---------|------------|--------------------|-------|
| <hr/> | | | |
| 1 | 0 | 0 | 1 |
| 2 | 1 | 2 | 4 |
| 4 | 2 | 8 | 16 |
| 8 | 3 | 24 | 64 |
| 16 | 4 | 64 | 256 |
| 32 | 5 | 160 | 1024 |
| 64 | 6 | 384 | 4096 |
| 128 | 7 | 896 | 16384 |
| 256 | 8 | 2048 | 65536 |
| \dots | | | |



Ordenación por selección



Ordenación por selección

Ordenación de arrays por selección

El array contiene inicialmente los datos desordenados:

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

A medida que se colocan los elementos en su lugar:

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

The diagram illustrates the state of an array during the selection sort process. The array has 10 slots, indexed from 0 to 9. The first four slots (0 to 3) contain the values 5, 7, 12, and 13, respectively. The remaining six slots (4 to 9) contain the values 20, 14, 27, 14, 32, and 15. A vertical dashed line is positioned at index 4, separating the ordered part from the part to be sorted. A yellow bracket below the first four slots is labeled "Parte ya ordenada: array[0...i-1]" and "todos son menores (o iguales) que los datos de la parte no ordenada". Another yellow bracket below the last six slots is labeled "Elementos por colocar: array[i...N-1]".

Parte ya ordenada: $\text{array}[0\dots i-1]$
todos son menores (o iguales) que
los datos de la parte no ordenada

Elementos por colocar:
 $\text{array}[i\dots N-1]$



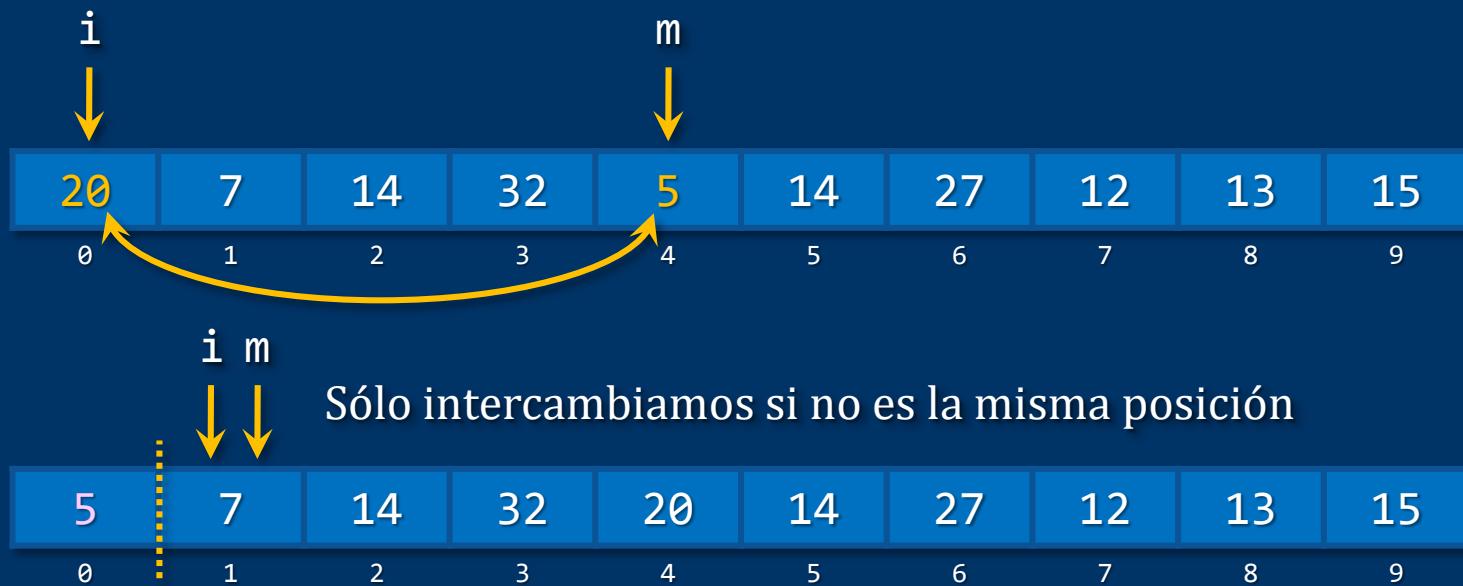
Ordenación por selección

Seleccionar el menor elemento de la parte no ordenada e intercambiar:

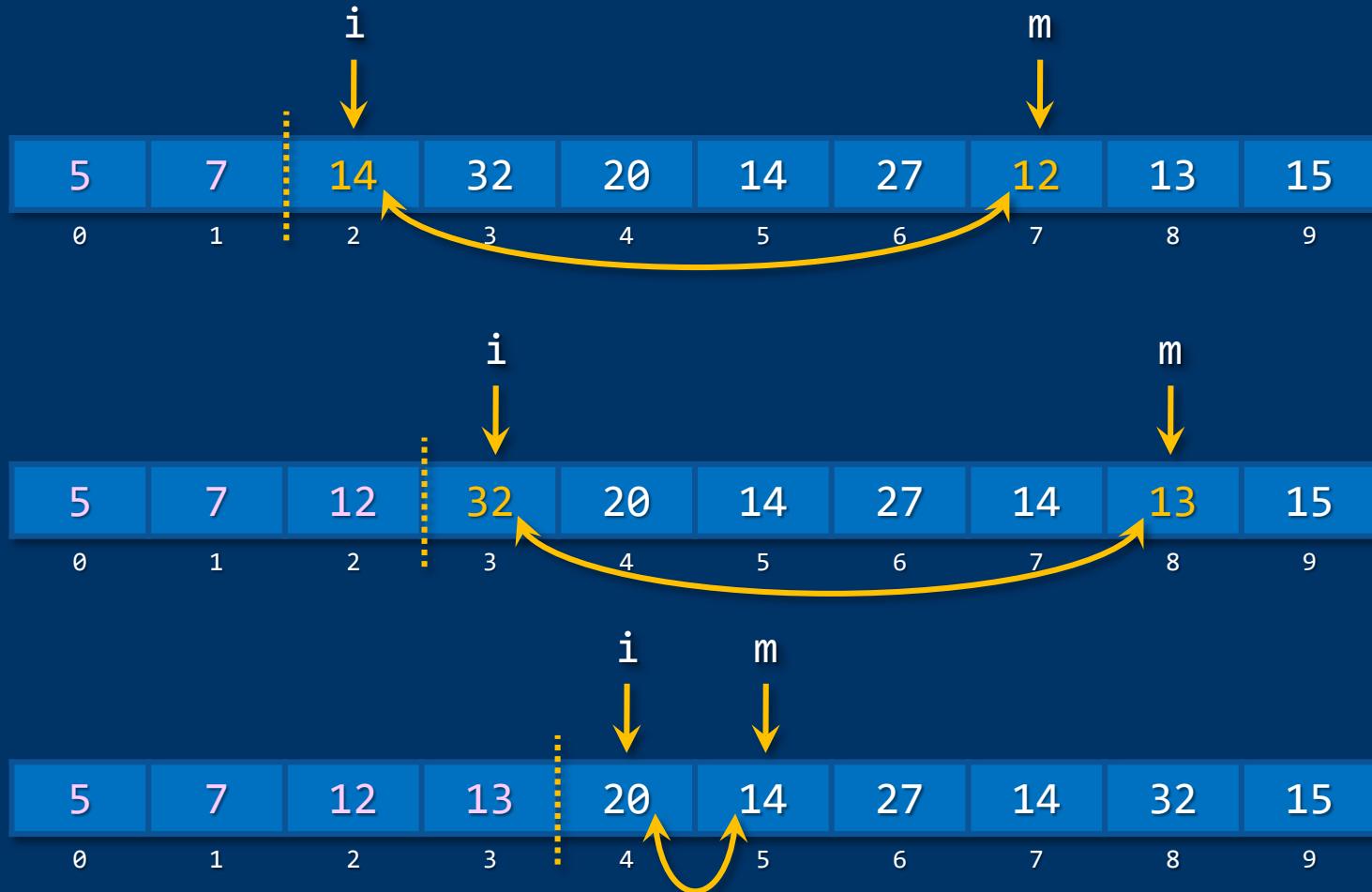
Desde la primera posición, $i = 0$, hasta la penúltima ($N-2$):

Posición m con el menor elemento desde $j = i$ hasta la última ($N-1$)

Intercambiar el elemento de la posición i con el de la posición m



Ordenación por selección



Ordenación por selección

```
// Tipo de datos de los elementos (con operadores: < y =)
typedef ... tData ... ;

void ordenarSeleccion(tData tabla[], int cont) {
    // ordenado tabla[0.. i-1] y todos los elementos de la parte
    // ordenada son menores (o iguales) que los de la parte no ordenada
    for (int i = 0; i < cont-1; ++i) { // hasta el penúltimo
        // colocar el menor de la parte no ordenada en tabla[i]
        // seleccionando el menor de la parte no ordenada (tabla[i..N-1])
        int menor = i;
        for (int j = i + 1; j < cont; ++j)
            if (tabla[j] < tabla[menor]) menor = j;
        if (i < menor) { // intercambiarlo con tabla[i]
            int tmp = tabla[i];
            tabla[i] = tabla[menor];
            tabla[menor] = tmp;
        }
    } // parte ordenada tabla[0 ... N-1]
}
```



Ordenación por selección

Complejidad

¿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 por ordenar:

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

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

Mismo número de comparaciones en todos los casos.

Complejidad cuadrática, igual que el método de inserción



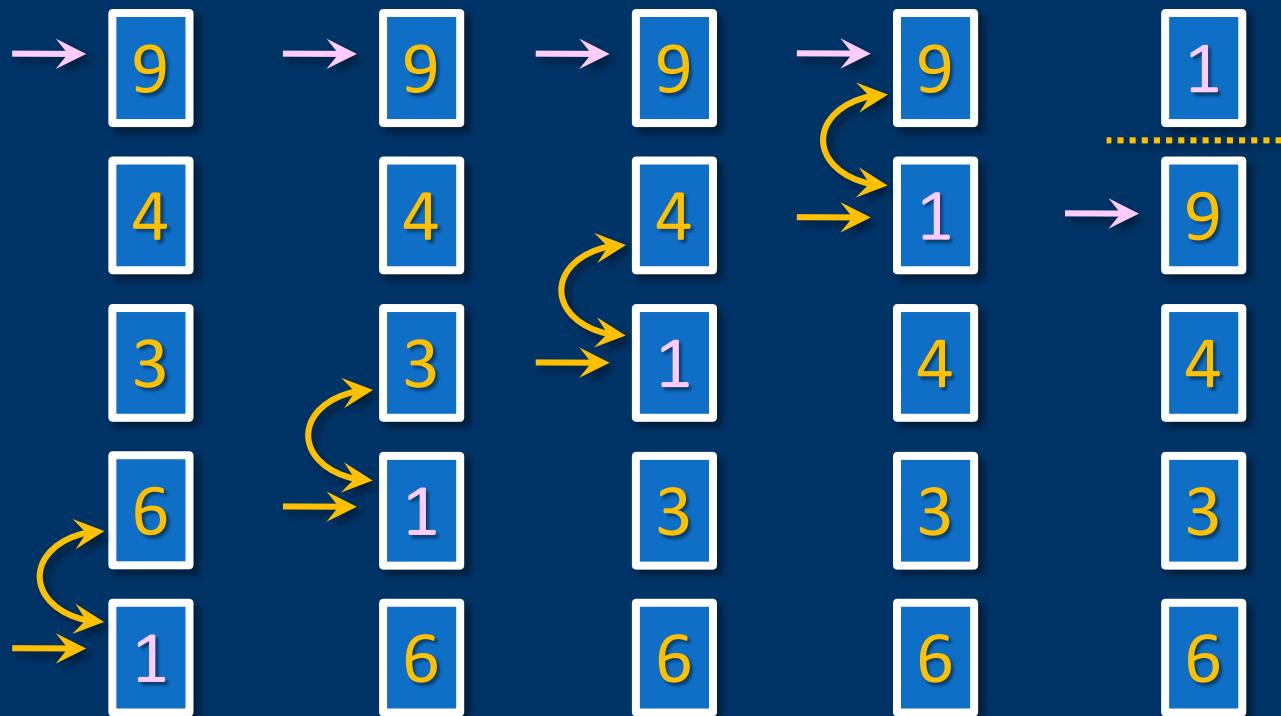
Método de la burbuja



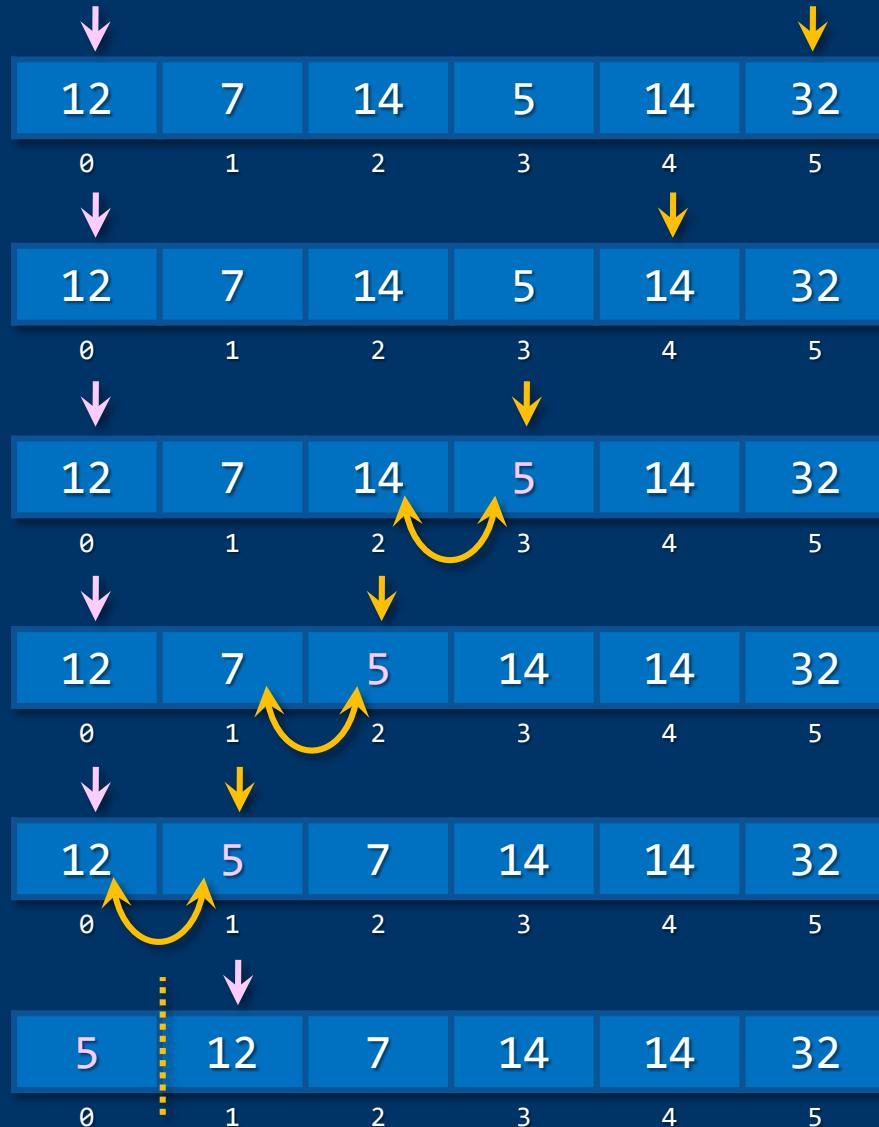
Método de la burbuja

Algoritmo de ordenación por el método de la burbuja

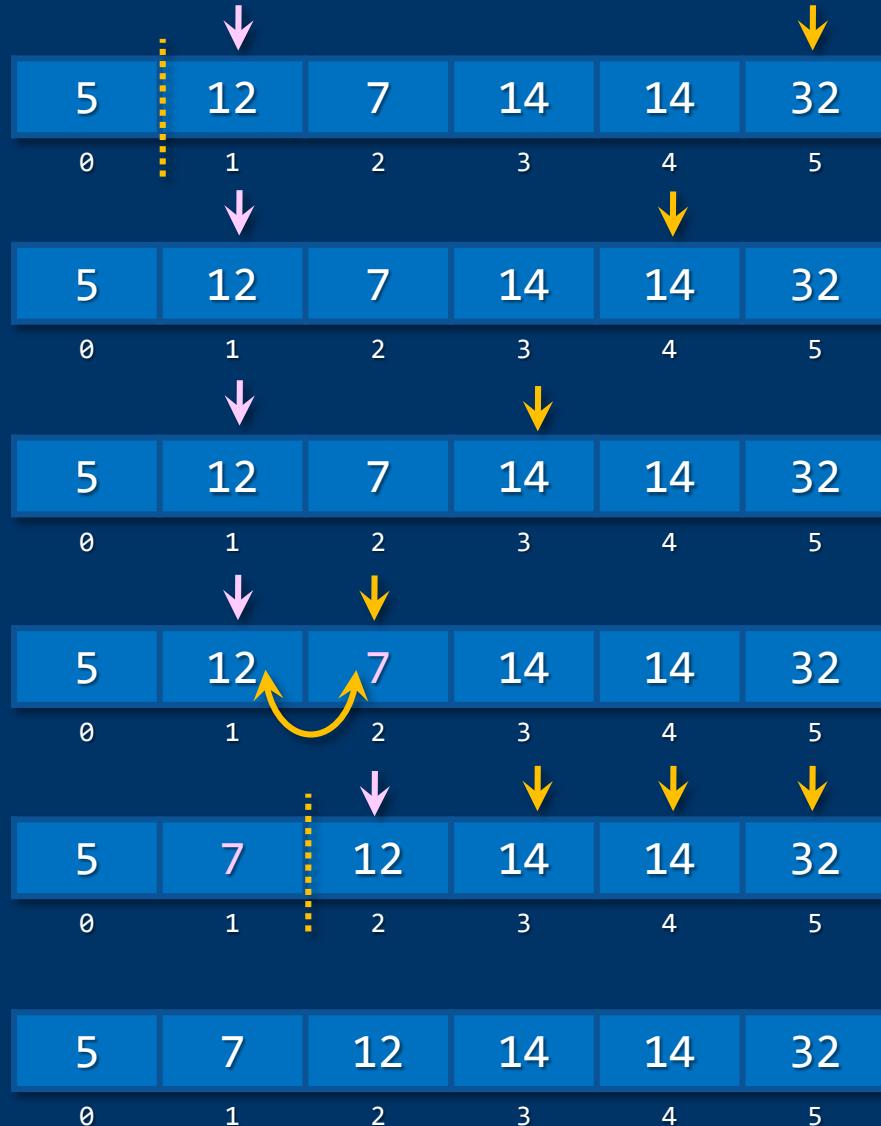
El elemento menor va *subiendo* hasta alcanzar su posición.



Método de la burbuja



Método de la burbuja



Método de la burbuja

Ordenación por el método de la burbuja (1)

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

```
// Tipo de datos de los elementos (con operadores: < y =)
typedef ... tDato ... ;

void ordenarBurbuja(tDato tabla[], int cont) {
    // ordenado tabla[0.. i-1] y todos los elementos de la parte ordenada son
    // menores (o iguales) que los de la parte no ordenada (tabla[i..N-1])
    for (int i = 0; i < cont-1; ++i) { // hasta el penúltimo
        // colocar el menor de la parte no ordenada en tabla[i]
        // intercambiando, desde el último, con los elementos mayores
        for (int j = cont-1; j > i; --j)
            if (tabla[j] < tabla[j-1])
                intercambiar(tabla[j], tabla[j-1]);
    }
}
```



Método de la burbuja

Algoritmo de ordenación por el método de la burbuja (1)

Complejidad de $O(N^2)$

Método de la burbuja

Si al colocar el i -ésimo menor no se ha realizado ningún intercambio, entonces la lista ya 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

$O(N^2)$

Si al colocar el i-ésimo menor no se ha realizado ningún intercambio, entonces la lista ya está ordenada y no es necesario seguir.

```
for (int i = 0; i < cont-1; ++i)
    // colocar el menor de la parte no ordenada en tabla[i] (i-ésimo menor)
    for (int j = cont-1; j > i; --j)
        if (tabla[j] < tabla[j-1])
            intercambiar(tabla[j], tabla[j-1]);
```



```
while (i < cont-1 && inter) {
    // colocar el menor de la parte no ordenada en tabla[i] (i-ésimo menor)
    inter = false; // comprobando si se realiza algún intercambio
    for (int j = cont-1; j > i; --j)
        if (tabla[j] < tabla[j-1]) {
            intercambiar(tabla[j], tabla[j-1]);
            inter = true;
        }
    ++i; }
```

Método de la burbuja

Método de la burbuja

$O(N^2)$

```
// Tipo de datos de los elementos (con operadores: < y =)
typedef ... tDatos ... ;

void ordenarBurbuja(tDatos tabla[], int cont) {
    bool inter = true;
    int i = 0;
    // ordenado tabla[0.. i-1] y todos los elementos de la parte
    // ordenada son menores (o iguales) que los de la parte no ordenada
    while (i < cont-1 && inter) { // hasta el penúltimo
        // colocar el menor de la parte no ordenada en tabla[i]
        // intercambiando, desde el último, con los elementos mayores
        inter = false; // y comprobando si se realiza algún intercambio
        for (int j = cont-1; j > i; --j)
            if (tabla[j] < tabla[j-1]) {
                intercambiar(tabla[j], tabla[j-1]);
                inter = true;
            }
        ++i;
    }
}
```

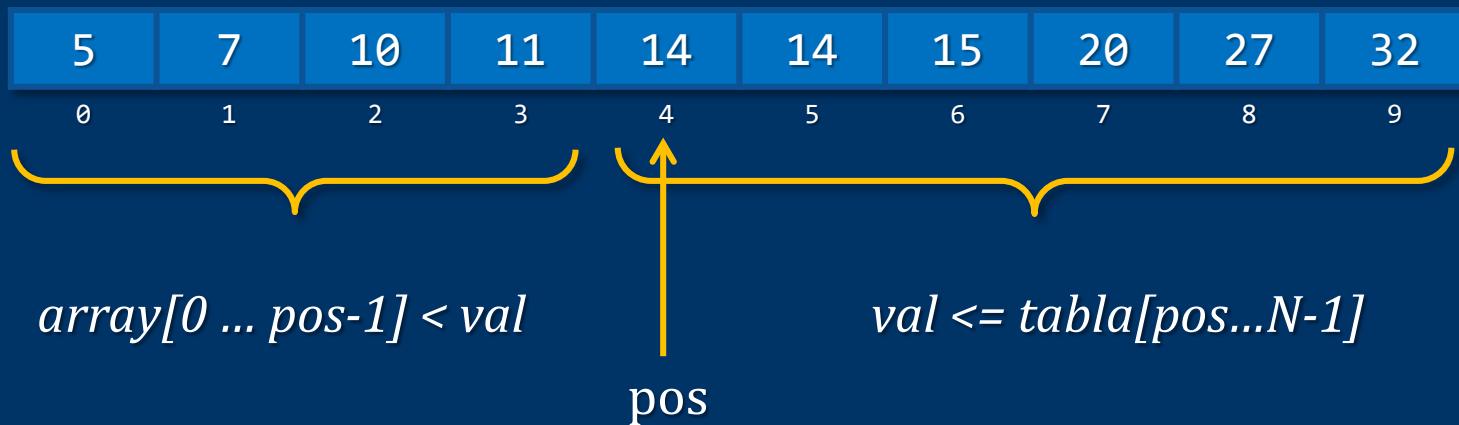


Búsquedas en arrays ordenados



Búsquedas en arrays ordenados

Búsqueda secuencial: dado un valor determinar la posición donde se encuentra o donde le correspondería estar en el orden



Si buscamos el 13, al llegar al 14 ya sabemos que no está, pues a continuación todos los valores son ≥ 14 y, por lo tanto > 13 .

Recorremos mientras que no lleguemos al final y el valor en la lista sea menor que el buscado.



Búsquedas en arrays ordenados

Búsqueda secuencial

Complejidad de $O(N)$

```
const int T = N;
typedef int tTabla[T];
                    // tabla ordenada
bool buscarSecuencial(const tTabla tabla, int buscado, int& pos)
{
    bool encontrado = false; pos = 0;
    // tabla[0 ... pos-1] < buscado
    while ((pos < T) && (tabla[pos] < buscado))
        ++pos;
    // pos < T -> buscado <= tabla[pos...T-1]
    if ((pos < T) && !(buscado < tabla[pos])))
        encontrado = true; // Encontrado, buscado == tabla[pos]

    return encontrado;
}
```



Búsquedas en arrays ordenados

Búsqueda binaria

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

Comparamos con el valor que esté en el medio del array.

Si es el que buscamos, terminamos.

Si no, si es mayor, buscamos en la primera mitad del array

Si no, si es menor, buscamos y en la segunda mitad del array

Repetimos el proceso hasta encontrarlo o quedarnos sin donde buscar.

Si buscamos el 12:

↓ Elemento mitad

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

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

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

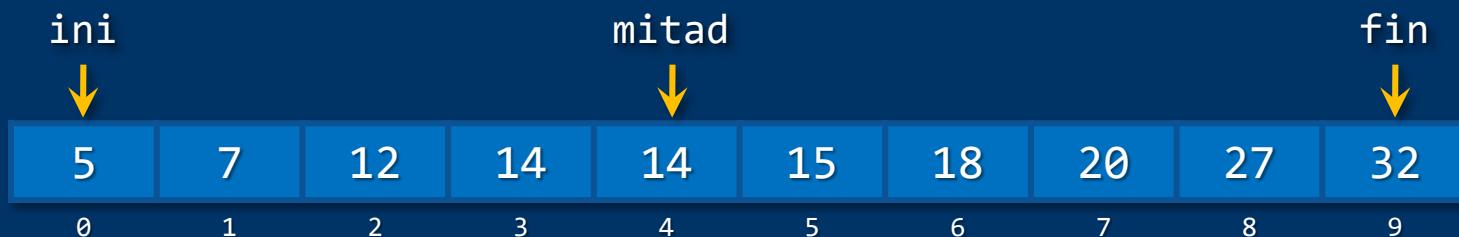


Búsqueda binaria



Búsqueda binaria

Vamos buscando en segmentos cada vez más pequeños (mitades). Delimitamos el segmento del array donde buscar en cada momento. Inicialmente tenemos todo el array:



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

Si no se ha encontrado, ¿dónde seguir buscando?

Si el buscado es menor que el que está en la mitad: $\text{fin} = \text{mitad} - 1$

Si el buscado es mayor que el que está en la mitad: $\text{ini} = \text{mitad} + 1$

Si $\text{fin} < \text{ini}$, no queda donde buscar

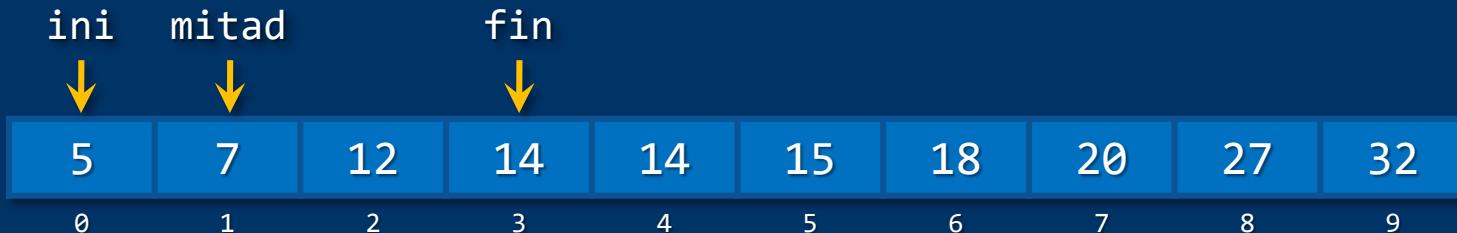


Búsqueda binaria

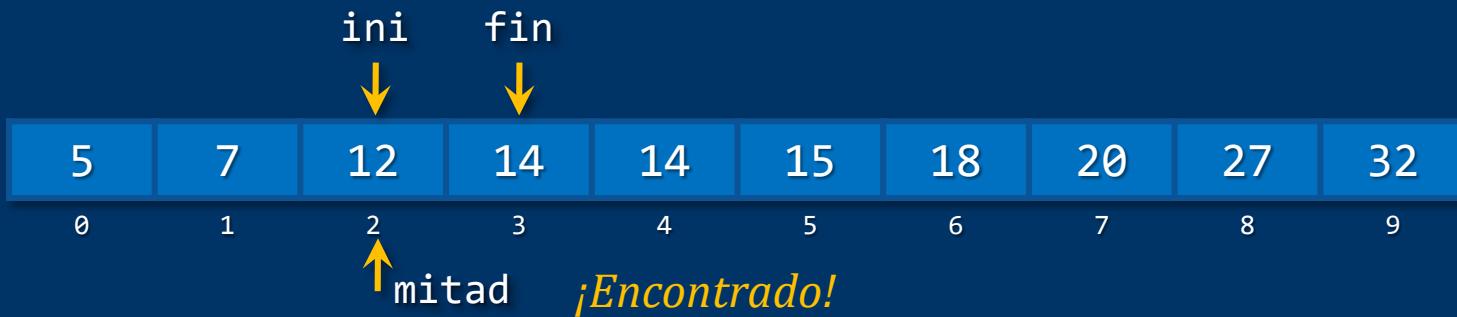
Buscamos el 12



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



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



Búsqueda binaria

Si el elemento no está, acabaremos quedándonos sin segmento: `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] → 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] → fin = mitad - 1

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



Búsqueda binaria

```
const int T = N;
typedef int tTabla[T];
// tabla ordenada
bool buscarBinaria(const tTabla tabla, int buscado, int& pos)
{
    int ini = 0, fin = T-1, mitad;
    bool encontrado = false;
    // tabla[0...ini-1] < buscado Y buscado < tabla[fin+1...N-1]
    // 0 <= ini <= N, -1 <= fin <= (N-1), ini<=(fin+1)
    while ((ini <= fin) && !encontrado) {
        mitad = (ini + fin) / 2; // División entera
        if (buscado < tabla[mitad]) fin = mitad - 1;
        else if (tabla[mitad] < buscado) ini = mitad + 1;
        else encontrado = true;
    }
    if (encontrado) pos = mitad; // en la posición mitad
    else pos = ini; // No encontrado, le corresponde
                    // la posición ini (=fin+1)
    return encontrado;
}
```



Búsqueda binaria

¿Qué orden de complejidad tiene?

Caso peor:

El elemento 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 , esa serie queda como:

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

De forma que el nº de elementos de esa serie es exactamente k .

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

Complejidad de $O(\log_2 N)$



Ejemplo

```
// lista.h

const int TM = ...;

typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tDato;

// typedef tDato tTabla[TM + 1]; // Possible centinela

typedef struct {
    tDato datos[TM + 1];
    int cont;
} tLista;

bool buscar(tLista const& lista, string const& nombre,
           int & pos);
```



Ejemplo

```
int main() {
    tLista lista;
    cargar(lista, "ordenados.txt");
    mostrar(lista);
    string nombre;
    int pos;
    cout << "Nombre: ";
    getLine(cin, buscado);
    if (buscar(lista, nombre, pos))
        cout << "Encontrado! ";
    else
        cout << "No encontrado! ";
    cout << "(posición: " << pos << ")" << endl;
    return 0;
}
```



Ejemplo

```
// lista.cpp

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

    return encontrado;
}
```



Operaciones con arrays ordenados



Operaciones con arrays ordenados

Inserciones y eliminaciones en listas ordenadas

Hay que insertar de forma que la lista siga ordenada:

Desplazar a la derecha desde el final, mientras que el nuevo elemento sea menor que el del array.

Por ejemplo, insertamos el 12:

¡Error si la lista está llena!

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



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

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



Operaciones con arrays ordenadas

Inserciones y eliminaciones en listas ordenadas

Al eliminar no hay que dejar huecos: Buscar el lugar (pos), si está, desplazar a la izquierda desde pos.

Por ejemplo, eliminamos el 7: Está en la posición 1



| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|----|
| 5 | 7 | 10 | 12 | 14 | 14 | 15 | 18 | 20 | 27 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|----|
| 5 | 7 | 10 | 12 | 14 | 14 | 15 | 18 | 20 | 27 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |



| | | | | | | | | | |
|---|----|----|----|----|----|----|----|----|---|
| 5 | 10 | 12 | 14 | 14 | 15 | 18 | 20 | 27 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |



Ejemplo

Gestión de una lista de datos ordenada por el campo nombre

```
// lista.h

const int TM = ...;

typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tDato;

// typedef tDato tTabla[TM + 1]; // Possible centinela

typedef struct {
    tDato datos[TM + 1];
    int cont;
} tLista;

...
```



Ejemplo

...

```
bool insertar(tLista & lista, tDato const& dato);  
  
bool eliminar(tLista & lista, int pos);  
bool eliminar(tLista & lista, string const& nombre);  
  
bool buscar(tLista const& lista, string const& nombre,  
            int & pos); // binaria
```



Ejemplo

```
int main() {
    tLista lista;  string nombre;
    ...
    else { ... int pos;
        cout << "Nombre: "; getline(cin, nombre);
        if(!buscar(lista, nombre, pos)){
            cout << "Nombre: " << nombre << "No existe" << endl;
            tDato dato; dato.nombre = nombre;
            cout << "Sueldo: "; cin >> dato.sueldo;
            cout << "Código: "; cin >> dato.codigo;
            if (insertar(lista, dato)) cout << "Insertado" << endl;
            else cout << "Error: Lista llena" << endl;
        }
        else {
            cout << "Nombre: " << nombre << endl;
            cout << "Sueldo: " << lista.datos[pos].sueldo << endl;
            cout << "Código: " << lista.datos[pos].codigo << endl;
        }
    ...
}
```

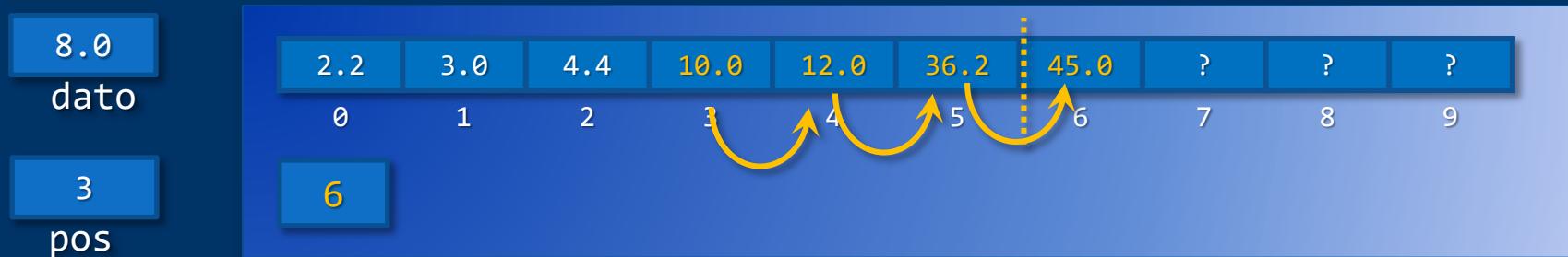


Operaciones con listas ordenadas



Operaciones con listas ordenadas

```
void desplazarDerecha(tLista & list, int pos){  
    // Abrir hueco  
    for (int i = list.cont; i > pos; --i)  
        list.datos[i] = list.datos[i - 1];  
}
```



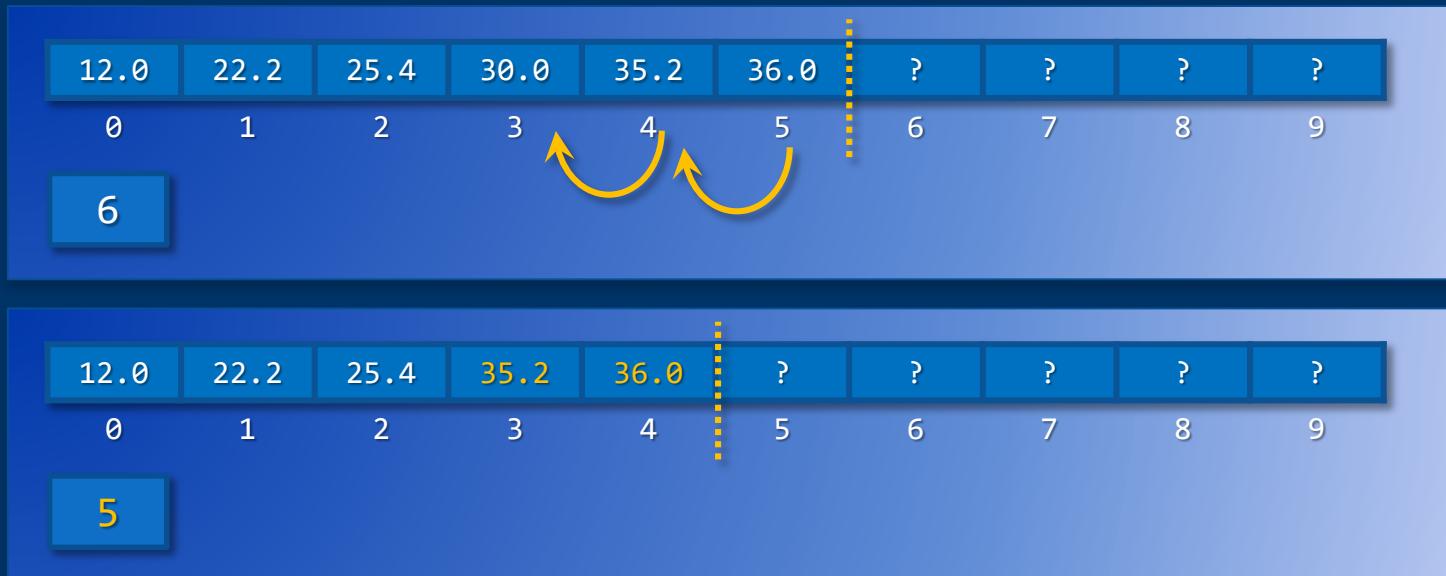
Operaciones con listas ordenadas

```
bool eliminar(tLista & lista, int pos){  
    if ((pos < 0) || (pos >= lista.cont))  
        return false; // Elemento inexistente  
    else {  
        desplazarIzquierda(lista, pos);  
        --lista.cont;  
        return true;  
    }  
}  
bool eliminar(tLista & lista, string const& nombre) {  
    int pos = 0;  
    if (buscar(lista, dato.nombre, pos))  
        return eliminar(lista, pos);  
    else return false;  
}
```



Operaciones con listas ordenadas

```
void desplazarIzquierda(tLista & list, int pos){  
    for (int i = pos + 1; i < list.cont ; ++i)  
        list.datos[i - 1] = list.datos[i];  
}
```



Operaciones con listas ordenadas

```
// insertar
bool insertar(tLista & lista, tDato const& dato) {
    if(lista.cont == TM) return false;
    else {
        // insertar ordenadamente: Desplazar a la derecha desde el
        // final, mientras el elemento del array sea mayor que el nuevo
        int pos = lista.cont;
        while(pos > 0 && dato < lista.datos[pos-1]) {
            lista.datos[pos] = lista.datos[pos-1];
            --pos;
        }
        lista.datos[pos] = dato;
        ++lista.cont;
        return true;
    }
}
```



Mezcla de arrays ordenados



Mezcla de arrays ordenados

Mezcla de dos listas ordenadas en arrays

```
const int TM = N;  
typedef struct {  
    int datos[TM];  
    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.



Mezcla de arrays ordenados

```
void mezcla(tLista const& lista1, tLista const& lista2,
            tLista & listaM) {
    int pos1 = 0, pos2 = 0;
    listaM.cont = 0;

    while ((pos1 < lista1.cont) && (pos2 < lista2.cont)
           && (listaM.cont < TM)) {

        if (lista1.datos[pos1] < lista2.datos[pos2]) {
            listaM.datos[listaM.cont] = lista1.datos[pos1];
            ++pos1;
        }
        else {
            listaM.datos[listaM.cont] = lista2.datos[pos2];
            ++pos2;
        }
        ++listaM.cont;
    }
    ...
}
```



Mezcla de arrays ordenados

...

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



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:

- ① **Reconocimiento (Attribution):**
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
- ② **No comercial (Non commercial):**
La explotación de la obra queda limitada a usos no comerciales.
- ③ **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.

