

TEMA 5

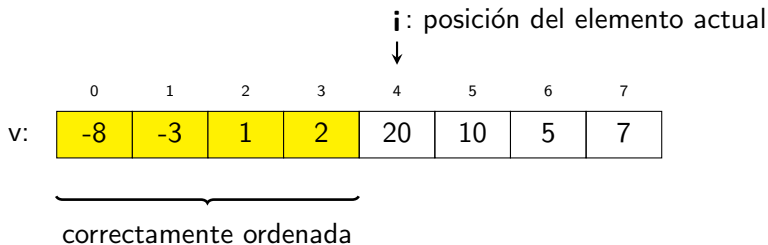
Algoritmos de ordenación

Emma Rollón
erollon@cs.upc.edu

Departamento de Ciencias de la Computación

- Algoritmos clásicos {
 - Selección
 - Inserción
 - Burbuja
 - Fusión
- Conexión con sort (biblioteca algorithm)
- Ordenar puede no ser buena idea!

Ordenación por selección (vector<int>)



Características:

- Parte tratada: correctamente ordenada según operador $<$
- Tratamiento elemento actual: Intercambiar $v[i]$ con el elemento menor de $v[i, \dots, v.size() - 1]$.

Ordenación por selección (vector<int>)

```
// Pre: a vale A, b vale B
// Post: a vale B, b vale A
void intercambiar(int& a, int& b) {
    int aux = a;
    a = b;
    b = aux;
}

// Pre: 0 <= from < v.size()
// Post: retorna la posición del elemento menor desde from hasta v.size() - 1
int pos_minim(const vector<int>& v, int from) {
    int p = from;
    for (int i = from + 1; i < v.size(); ++i) {
        if (v[i] < v[p]) p = i; // el operador < tiene que estar definido!
    }
    return p;
}

// Pre: v es válido
// Post: v queda ordenado ascendentemente según operador <
void ordenacion_seleccion(vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n - 1; ++i) {
        int p_min = pos_minim(v, i);
        intercambiar(v[i], v[p_min]);
    }
}
```

Ordenación por selección (vector<T>)

Y si el tipo de datos T de los elementos del vector no tienen definido el operador < que indica la ordenación que quiero?

- Cuando el operador < se utiliza en un algoritmo de ordenación, **a menor que b** significa que **a irá antes que b** una vez el vector esté ordenado.
- Fíjate que el operador < lo podríamos implementar como una función:

```
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(int a, int b) {
    return a < b;
}
```

- Cuando ordenamos un vector<T>, también podemos implementar una función similar, con el significado menor que queramos:

```
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(const T& a, const T& b) {
    ...
}
```

Ordenación por selección (vector<T>)

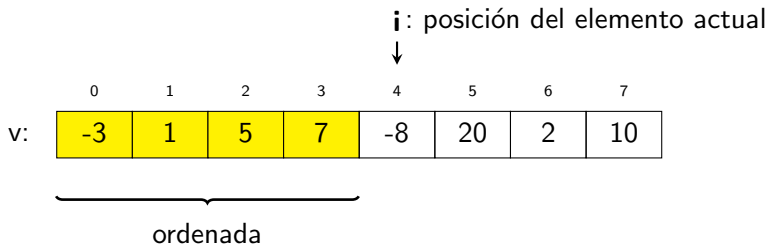
```
// Pre: -
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(const T& a, const T& b) {
    ...
}

// Pre: a vale A, b vale B
// Post: a vale B, b vale A
void intercambiar(T& a, T& b) {
    ...
}

// Pre: 0 <= from < v.size()
// Post: retorna la posición del elemento menor desde from hasta v.size() - 1
int pos_minim(const vector<T>& v, int from) {
    int p = from;
    for (int i = from + 1; i < v.size(); ++i) {
        if (operador_menor(v[i], v[p])) p = i;    // función que implementa comparación
    }
    return p;
}

// Pre: v es válido
// Post: v queda ordenado ascendentemente según operador_menor
void ordenacion_seleccion(vector<T>& v) {
    int n = v.size();
    for (int i = 0; i < n - 1; ++i) {
        int p_min = pos_minim(v, i);
        intercambiar(v[i], v[p_min]);
    }
}
```

Ordenación por inserción ($\text{vector}<\text{int}>$)



Características:

- Parte tratada: ordenada según operador $<$
- Tratamiento elemento actual: Llevar elemento $v[i]$ a su posición correcta en $v[0, \dots, i]$, que quedará ordenado.

Ordenación por inserción (vector<int>)

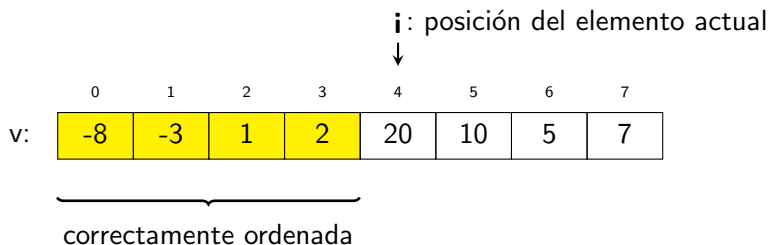
```
// Pre: v es válido
// Post: v queda ordenado ascendentemente según operador <
void ordenacion_insercion(vector<int>& v) {
    int n = v.size();
    for (int i = 1; i < n; ++i) {
        // buscar la posición correcta desde i hasta 0 para el elemento v[i]
        int x = v[i];
        int j = i;
        while (j > 0 and v[j - 1] > v[j]) {
            v[j] = v[j - 1];
            --j;
        }
        // al salir del bucle, j es la posición que buscábamos
        v[j] = x;
    }
}
```


Ordenación por inserción (vector<T>)

```
// Pre: —
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(const T& a, const T& b) {
    ...
}

// Pre: v es válido
// Post: v queda ordenado ascendentemente según operador_menor
void ordenacion_insercion(vector<T>& v) {
    int n = v.size();
    for (int i = 1; i < n; ++i) {
        // buscar la posición correcta desde i hasta 0 para el elemento v[i]
        int x = v[i];
        int j = i;
        while (j > 0 and operador_menor(v[j], v[j - 1])) {
            v[j] = v[j - 1];
            —j;
        }
        // al salir del bucle, j es la posición que buscábamos
        v[j] = x;
    }
}
```

Ordenación de la burbuja (vector<int>)



Características:

- Parte tratada: correctamente ordenada según operador $<$
- Tratamiento elemento actual: Llevar a la posición i el elemento menor de $v[i, \dots, v.size() - 1]$ comparando dos a dos los elementos desde el final del vector hasta la posición i .

Ordenación de la burbuja (vector<int>)

```
// Pre: a vale A, b vale B
// Post: a vale B, b vale A
void intercambiar(int& a, int& b) {
    int aux = a;
    a = b;
    b = aux;
}

// Pre: v es válido
// Post: v queda ordenado ascendentemente según operador <
void ordenacion_burbuja(vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n - 1; ++i) {
        // llevar a v[i] el elemento menor en v[i, ..., n-1]
        // dejando eso elementos parcialmente ordenados
        for (int j = n - 1; j > i; --j) {
            if (v[j - 1] > v[j]) intercambiar(v[j - 1], v[j]);
        }
    }
}
```

Mejoras:

- Si en todo un pase del bucle interno no se produce ningún intercambio, entonces todo el vector ya está ordenado.
- Avanzar i hasta la posición del último intercambio.

Ordenación de la burbuja (vector<T>)

```
// Pre: —
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(const T& a, const T& b) {
    ...
}

// Pre: a vale A, b vale B
// Post: a vale B, b vale A
void intercambiar(T& a, T& b) {
    int aux = a;
    a = b;
    b = aux;
}

// Pre: v es válido
// Post: v queda ordenado ascendentemente según operador_menor
void ordenacion_burbuja(vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n - 1; ++i) {
        // llevar a v[i] el elemento menor en v[i, ..., n-1]
        // dejando eso elementos parcialmente ordenados
        for (int j = n - 1; j > i; --j) {
            if (operador_menor(v[j], v[j - 1]) intercambiar(v[j - 1], v[j]);
        }
    }
}
```

Ordenación por fusión (vector<int>)

```
// Pre: v válido, 0 <= esq <= m < dre < v.size()
// Post: v[esq ... dre] queda ordenado según operador <
void fusion(vector<int>& v, int esq, int m, int dre) {
    int n = dre - esq + 1;
    vector<int> aux(n);
    int k = 0; // índice sobre aux
    int i = esq;
    int j = m + 1;
    while (i <= m and j <= dre) {
        if (v[i] < v[j]) { aux[k] = v[i]; ++i; }
        else { aux[k] = v[j]; ++j; }
        ++k;
    }
    while (i <= m) { aux[k] = v[i]; ++i; ++k; }
    while (j <= dre) { aux[k] = v[j]; ++j; ++k; }
    for (k = 0; k < n; ++k) v[esq + k] = aux[k];
}

// Pre: v es válido, 0 <= esq <= dre < v.size()
// Post: v[esq ... dre] queda ordenado ascendentemente según operador <
void ordenacion_fusion(vector<int>& v, int esq, int dre) {
    if (esq < dre) {
        int m = (esq + dre)/2;
        ordenacion_fusion(v, esq, m);
        ordenacion_fusion(v, m + 1, dre);
        fusion(v, esq, m, dre);
    }
}

int main() {
    ....
    ordenacion_fusion(v, 0, v.size() - 1);
}
```

Ordenación por fusión (vector<T>)

```
// Pre: -
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(const T& a, const T& b) { ... }

// Pre: v válido, 0 <= esq <= m < dre < v.size()
// Post: v[esq ... dre] queda ordenado según operador_menor
void fusion(vector<T>& v, int esq, int m, int dre) {
    int n = dre - esq + 1;
    vector<T> aux(n);
    int k = 0; // índice sobre aux
    int i = esq;
    int j = m + 1;
    while (i <= m and j <= dre) {
        if (operador_menor(v[i], v[j])) { aux[k] = v[i]; ++i; }
        else { aux[k] = v[j]; ++j; }
        ++k;
    }
    while (i <= m) {aux[k] = v[i]; ++i; ++k;}
    while (j <= dre) {aux[k] = v[j]; ++j; ++k;}
    for (k = 0; k < n; ++k) v[esq + k] = aux[k];
}

// Pre: v es válido, 0 <= esq <= dre < v.size()
// Post: v[esq ... dre] queda ordenado ascendentemente según operador_menor
void ordenacion_fusion(vector<T>& v, int esq, int dre) {
    if (esq < dre) {
        int m = (esq + dre)/2;
        ordenacion_fusion(v, esq, m);
        ordenacion_fusion(v, m + 1, dre);
        fusion(v, esq, m, dre);
    }
}
```

Conexión con sort (biblioteca algorithm)

En el tema de vectores vimos que:

```
// Post: retorna true si a tiene que ir antes que b en el vector ,  
//       false en caso contrario  
bool nom_func_bool(const T& a, const T& b) {  
    ...  
}  
  
int main() {  
    vector<T> v;  
    ....  
    sort(v.begin(), v.end(), nom_func_bool);  
    ....  
}
```

Sort implementa un algoritmo de ordenación diferente a los que hemos visto. Sin embargo:

La función *nom_func_bool* es el *operador_menor* que hemos visto ahora.

Conexión con sort (biblioteca algorithm)

Dado un vector de enteros, ordénalo de manera descendente.

```
// Post: retorna true si a es "menor" que b según criterio
//          descendente, false en caso contrario
bool operador_menor(int a, int b) {
    return a > b;
}

int main() {
    vector<int> v;
    ....
    sort(v.begin(), v.end(), operador_menor);
    ....
}
```


Conexión con sort (biblioteca algorithm)

Dado un vector de enteros, ordénalo de manera que los múltiplos de 3 aparezcan antes que el resto de números. Para la parte del vector con los múltiplos de 3 queremos que esté ordenada de forma ascendente, para la parte que no son múltiplos de 3 queremos que esté ordenada de forma descendente.

```
// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(int a, int b) {
    mult_a = (a%3 == 0);
    mult_b = (b%3 == 0);
    if (mult_a != mult_b) return mult_a;
    if (mult_a) return a < b; // Los dos son múltiplos de 3
    return a > b; // Los dos no son múltiplos de 3
}

int main() {
    vector<int> v;
    ....
    sort(v.begin(), v.end(), operador_menor);
    ....
}
```

Conexión con sort (biblioteca algorithm)

Dado un vector de Persona, ordénalo por edad ascendente, para los que tengan misma edad por nombre descendente, y para los que también tengan el mismo nombre, por dni ascendente.

```
struct Persona {
    string nombre;
    int edad;
    int dni;
};

// Post: retorna true si a es menor que b, false en caso contrario
bool operador_menor(const Persona& a, const Persona& b) {
    if (a.edad != b.edad) return a.edad < b.edad;
    if (a.nombre != b.nombre) return a.nombre > b.nombre;
    return a.dni < b.dni;
}

int main() {
    vector<Persona> v;
    ....
    sort(v.begin(), v.end(), operador_menor);
    ....
}
```

Ordenar puede no ser buena idea!

Ordenar es una tarea **costosa** (tiene un **doble bucle anidado!**).

Hay tareas que **no** se tienen que resolver **ordenando**. Ejemplos:

- Encontrar el **mínimo** o el **máximo**: haciendo un recorrido del vector es suficiente (un único bucle).
- **Contar** cuántos elementos cumplen una condición: haciendo un recorrido del vector es suficiente (un único bucle).

Siempre piensa que si lo puedes solucionar con un **único bucle**, entonces (en general) será **mejor que ordenando** el vector (a pesar de que visualmente utilices con sort menos líneas de código).