Un vector multidimensional es un vector cuyos elementos son, a la vez, un vector.

Por ejemplo:

- un vector cuyos elementos son vectores de int sería:

```
vector <vector <int> >              // hay 2 dimensiones
// |         |
// |         +--- segunda dimensión o vector interno
// +------ primera dimensión o vector externo
```

- un vector cuyos elementos son vectores cuyos elementos son vectores de int sería:

```
vector <vector <vector <int> > >  // hay 3 dimensiones
```

Cada vez que utilizamos la palabra vector decimos que tenemos una nueva dimensión.

Declaración de una matriz

Sintaxis:

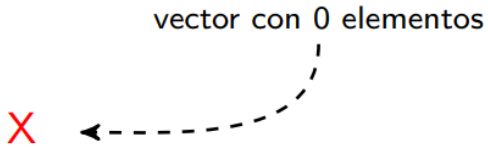
```
#include <vector>          // Biblioteca necesaria

vector< vector <T> > nombre_matriz(S, I);
```

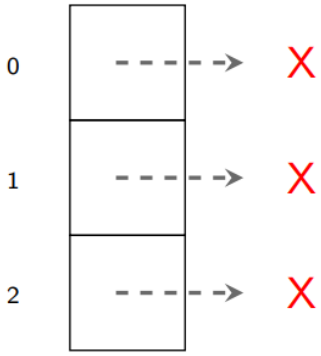
- T: tipo de datos simple
- S: tamaño del vector externo (por defecto es 0)
- I: inicialización del vector interno (si no se indica será un vector con 0 elementos)

Los parámetros S, I son opcionales:

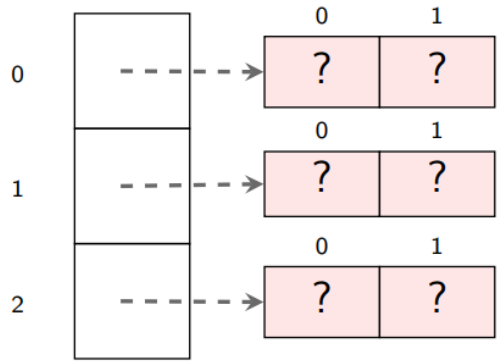
- Si sólo indicas uno, será S.
- Si indicas dos, serán S, I.



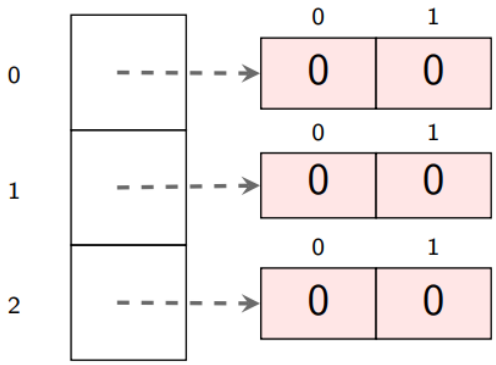
vector <vector <int>> mat;



vector <vector <int>> mat(3);



vector <vector <int>> mat(3, vector<int>(2));



vector<vector<int>> mat(3, vector<int>(2, 0));

Declaración de una matriz: simplificación

Vamos a definir alias para los vectores que representan filas y para el vector que representa la matriz gracias a la instrucción typedef.

```
typedef vector <int> Fila;
typedef vector <Fila> Matriz;
```

Las declaraciones anteriores quedarían como:

- Matriz con 0 filas y 0 columnas:

```
Matriz mat;
```

- Matriz con 3 filas con 0 columnas cada una:

```
Matriz mat(3);
```

- Matriz con 3 filas con 2 columnas cada una en donde el valor de los elementos se desconoce:

```
Matriz mat(3, Fila(2));
```

- Matriz con 3 filas con 2 columnas cada una en donde el valor de todos los elementos es 0:

```
Matriz mat(3, Fila(2, 0));
```

Uso de matrices: acceso

Siempre se accede por índice, en cada una de sus dimensiones:

```
nombre_matriz[E1][E2]
```

- E1, E2: expresión entera.
- Se accede a la posición E1 del vector externo nombre_matriz (fila E1) y a la posición E2 del correspondiente vector interno (columna E2). Sirve tanto para consultar como para actualizar.

¡Alerta!

Si se accede a una posición no válida: container with out-of-bounds index.

```
int main() {
    vector <vector <int> > mat(3, vector <int> (2, 1));
    vector <int> v = mat[0];      // copio un vector interno en v
    int x = mat[2][0];            // obtengo un elemento del vector interno
    mat[2][0] = 10;               // actualizo elemento del vector interno
    cout << x << " " << mat[2][0] << endl;
}
```

Uso de matrices: saber número de elementos

Número de elementos del vector externo:

```
int n = nombre_matriz.size();
```

- Nos dará el número de filas de la matriz.

Número de elementos de algún vector interno:

```
int m = nombre_matriz[índice].size();
```

- Nos dará el número de columnas de la matriz.
- El índice tiene que ser válido.

```
int main() {
    vector <vector <int> > mat(3, vector <int> (2, 1));
    int n = mat.size();
    int m = mat[0].size();
    cout << n << " " << m << endl;     // escribe 3 2
}
```

```
vector <vector <int> > mat;
cin >> mat;     // ERROR de compilación
cout << mat;    // ERROR de compilación
```

Recuerda que cin/cout sólo están definidos sobre tipos de datos simples.

¿Entonces cómo leemos y escribimos una matriz?

Lo haremos haciendo un recorrido en donde visitaremos todos los elementos de la matriz y los iremos leyendo/escribiendo. Vamos a distinguir dos casos:

- Leer/escribir matriz rectangular (todas las filas tienen el mismo número de columnas).
- Leer/escribir matriz irregular (cada fila tiene un número determinado de columnas).

Uso de matrices: lectura/escritura matriz rectangular

```
// Pre: en la entrada nos dan dos enteros que son número de filas f y columnas c
//      de la matriz. A continuación nos dan f líneas, cada una con c enteros
// Post: lee la matriz que se nos da en la entrada, hace una cierta tarea, y la escribe.
int main() {
    int f, c;
    cin >> f >> c;

    // crear matriz
    vector <vector <int> > mat(f, vector <int>(c));

    // inicializar la matriz con los valores de la entrada
    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) cin >> mat[i][j];
    }

    // se realiza la tarea correspondiente
    // ...

    // escribir el contenido de la matriz
    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) cout << mat[i][j] << " ";
        cout << endl;
    }
}
```

Uso de matrices: lectura/escritura matriz irregular

```
// Pre: en la entrada nos dan un entero que son número de filas f. A continuación
//      nos dan f líneas, cada una empieza con un número c que es el número de
//      columnas de esa fila y, a continuación, c enteros que son los
//      elementos de esa fila.
// Post: lee la matriz que se nos da en la entrada, hace una cierta tarea, y la escribe.
int main() {
    int f;
    cin >> f;

    // crear matriz
    vector <vector <int> > mat(f);

    // inicializar la matriz con los valores de la entrada
    for (int i = 0; i < f; ++i) {
        int c;
        cin >> c;
        mat[i] = vector <int>(c);
        for (int j = 0; j < c; ++j) cin >> mat[i][j];
    }

    // se realiza la tarea correspondiente
    // ...

    // escribir el contenido de la matriz
    for (int i = 0; i < f; ++i) {
        int c = mat[i].size();
        for (int j = 0; j < c; ++j) cout << mat[i][j] << " ";
        cout << endl;
    }
}
```

Paso de parámetros

Por valor:

- Todo lo que se ha explicado para un vector unidimensional aplica para los multidimensionales. Es decir, utilizaremos un paso por valor simulado: paso por referencia constante.
- Nada nuevo.

```
int main() {
    Matriz mat(3, Fila(4, 1));
    mi_accion(mat);
}
```

```
void mi_accion(const Matriz& m) {
    ...
}
```

Por referencia:

- Todo lo que se ha explicado para un vector unidimensional aplica para los multidimensionales.
- Nada nuevo.

```
int main() {
    Matriz mat(3, Fila(4, 1));
    mi_accion(mat);
}
```

```
void mi_accion(Matriz& m) {
    ...
    ... m[i][j] = 10;
    ...
}
```

Paso de parámetros: por referencia constante (ejemplo)

Acción que escribe el contenido de una matriz rectangular/irregular:

```
typedef vector<int> Fila;
typedef vector<Fila> Matriz;

// Pre: m es válida, rectangular y no es vacía
// Post: escribe el contenido de m
void escribir_matriz(const Matriz& m) {
    int f = m.size();
    int c = m[0].size();      // el acceso a m[0] es correcto porque m no es vacía
    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) cout << " " << m[i][j];
        cout << endl;
    }
}

// Pre: en la entrada nos dan los datos necesarios para inicializar una matriz de enteros
// Post: lee la matriz que se nos da en la entrada, hace una cierta tarea, y la escribe.
int main() {
    int f, c;
    cin >> f >> c;
    Matriz mat(f, Fila(c));
    // lectura de la matriz
    ...
    // tarea sobre la matriz
    ...
    // escritura de la matriz
    escribir_matriz(mat);
}
```

Paso de parámetros: por referencia (ejemplo)

La lectura de una matriz rectangular/irregular se puede encapsular como una acción. A continuación te damos una opción posible:

```
typedef vector<int> Fila;
typedef vector<Fila> Matriz;

// Pre: m es rectangular y no es vacía;
//   en la entrada nos dan los elementos fila a fila de m
// Post: m está inicializada con los datos de la entrada
void leer_matriz(Matriz& m) {
    int f = m.size();
    int c = m[0].size();      // el acceso a m[0] es correcto porque m no es vacía
    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) cin >> m[i][j];
    }
}

// Pre: en la entrada nos dan dos enteros que son número de filas f (mayor a 0)
//    y columnas c de la matriz. A continuación nos dan f líneas, cada una con c enteros
// Post: lee la matriz que se nos da en la entrada, hace una cierta tarea, y la escribe.
int main() {
    int f, c;
    cin >> f >> c;
    Matriz mat(f, Fila(c));
    leer_matriz(mat);
    // ...
}
```

Retorno de una función

Una función puede retornar un valor de cualquier tipo de datos. En particular, puede retornar un vector multidimensional.

Ejemplo: la lectura de una matriz rectangular/irregular se puede encapsular como una función. A continuación te damos una opción posible:

```
// Pre: en la entrada nos dan dos enteros que son número de filas f (mayor a 0)
//    y columnas c de la matriz. A continuación nos dan f líneas, cada una con c enteros
// Post: m está inicializada con los datos de la entrada
Matriz leer_matriz() {
    int f, c;
    cin >> f >> c;
    Matriz m(f, Fila(c));
    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) cin >> m[i][j];
    }
    return m;
}

// Pre: en la entrada nos dan dos enteros que son número de filas f (mayor a 0)
//    y columnas c de la matriz. A continuación nos dan f líneas, cada una con c enteros
// Post: lee la matriz que se nos da en la entrada, hace una cierta tarea, y la escribe.
int main() {
    Matriz mat = leer_matriz();
    // ...
}
```

Sumar uno a todos los elementos de una matriz

$$\begin{pmatrix} 3 & 2 & 4 \\ 1 & 6 & 8 \end{pmatrix} \xrightarrow{\text{sumar 1 a todos los elementos}} \begin{pmatrix} 4 & 3 & 5 \\ 2 & 7 & 9 \end{pmatrix}$$

Solución 1: Recorrer la matriz por filas

- Por cada valor i que corresponde con un índice válido de fila:
- Por cada valor j que corresponde con un índice válido de columna:
- Sumar 1 al elemento mat[i][j]

Código:

```
// Pre: mat es no vacía y válida
// Post: a los elementos originales de mat se ha sumado 1
void suma_uno(Matriz& mat) {
    int n = mat.size();
    int m = mat[0].size();        // acceso correcto porque mat no es vacía
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <  m; ++j) ++mat[i][j];
    }
}
```

Suma de matrices

Pasos:

- f y c son el número de filas y columnas respectivamente que tendrá la matriz resultado
- Declaramos la matriz resultado suma
- Por cada valor i de 0 a f - 1 (índice válido de fila en suma):
- Por cada valor j de 0 a c - 1 (índice válido de columna en suma):
- Inicializar suma[i][j] con mat1[i][j] + mat2[i][j]
- Retornar suma

```
typedef vector <vector <int> > Matriz;

// Pre: mat1, mat2 matrices no vacías y válidas con las mismas dimensiones
// Post: retorna mat1 + mat2
Matriz suma_matrices(const Matriz& mat1, const Matriz& mat2) {
    int f = mat1.size();
    int c = mat1[0].size();        // acceso correcto porque mat1 es no vacía
    Matriz suma(f, vector<int>(c));
    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) suma[i][j] = mat1[i][j] + mat2[i][j];
    }
    return suma
}
```

Matriz transpuesta

```
typedef vector <vector <int> > Matriz;

// Pre: mat es una matriz rectangular no vacía y válida
// Post: retorna la matriz transpuesta de mat
Matriz transpuesta(const Matriz& mat) {
    int f  = mat.size();
    int c = mat[0].size();      // correcto (mat no es vacía)
    Matriz t (c, vector <int> (f));
    for (int j = 0; j < c; ++j) {
        for (int i = 0; i < f; ++i) t[j][i] = mat[i][j];
    }
    return t;
}
```

Fíjate que:

- i está asociado a la primera dimensión de mat, pero a la segunda en t
- j está asociado a la segunda dimensión de mat, pero a la primera en t

Matriz simétrica

```
// Pre: mat es una matriz cuadrada válida
// Post: retorna true si la matriz es simétrica, false si no.
bool simetrica(const Matriz& mat) {
    int n = mat.size();    // es una matriz de n*n (cuadrada)
    for (int i = 0; i < n - 1; ++i) {
        for (int j = i + 1; j <  n; ++j) {
            if (mat[i][j] != mat[j][i]) return false;
        }
    }
    return true;
}
```

Producto de matrices

```
// Pre: A, B matrices no vacías con dimensiones n*p y p*q
// Post: A * B que tendrá dimensión n*q
Matriz producto_matrices(const Matriz& A, const Matriz& B) {
    int n = A.size();
    int p = B.size();        // equivalente: int p = A[0].size();
    int q = B[0].size();
    Matriz C(n, vector <int>(q, 0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < q; ++j) {
            // calcular el valor del elemento C[i][j]
            // en la declaración lo hemos inicializado a 0
            for (int k = 0; k < p; ++k) {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
    return C;
```

Search in a matrix

```
// Pre:  m is a non-empty matrix
// Post: i and j define the location of a cell
//       that contains the value x in M.
//       In case x is not in m, then i = j = -1

void search(const Matrix& m, int x, int& i, int& j) {
    int nrows = m.size();
    int ncols = m[0].size();
    bool found = false;
    int i = 0;
    while (not found and i < nrows) {
        int j = 0;
        while (not found and j < ncols) {
            if (m[i][j] == x) found = true;
            ++j;
        }
        ++i;
    }
    if (not found) {
        i = -1;
        j = -1;
    }
}
```

Search in a sorted matrix

```
// Pre:  m is non-empty and sorted by rows and columns
//       in ascending order.
// Post: i and j define the location of a cell that contains the value
//       x in m. In case x is not in m, then i=j=-1.

void search(const Matrix& m, int x, int& i, int& j) {
    int nrows = m.size();
    int ncols = m[0].size();
    i = nrows - 1;
    j = 0;
    bool found = false;
    // Invariant: x can only be found in M[0..i,j..ncols-1]
    while (not found and i >= 0 and j < ncols) {
        if (m[i][j] < x) ++j;
        else if (m[i][j] > x) --i;
        else found = true;
    }

    if (not found) {
        i = -1;
        j = -1;
    }
```

Una tupla es un tipo de datos que nos permite agrupar bajo un mismo nombre un conjunto fijo de valores con (posiblemente) diferentes tipos de datos.

$$\text{Reloj} \begin{cases} \text{hora (int)} \\ \text{min (int)} \\ \text{sec (int)} \end{cases} \qquad \text{Racional} \begin{cases} \text{num (int)} \\ \text{den (int)} \end{cases} \qquad \text{Film} \begin{cases} \text{title (string)} \\ \text{year (int)} \end{cases}$$

Cada valor de la tupla se denomina campo y tiene:

- un nombre (siempre en minúscula)
- un tipo de datos (cualquiera de los que hemos visto)

Sintaxis:

```
...
using namespace std;

struct Nombre_tupla {
    tipo_de_datos campo1;
    ...;
    tipo_de_datos campoN;
};                      // <------- acaba con un ;

int main() {
    Nombre_tupla nombre_var;
    ...
}
```

Semántica:
Se crea el tipo de datos llamado Nombre_tupla. A partir de ese momento se pueden declarar variables de ese tipo de datos.

Observación

Nombre_tupla no tiene definido ninguno de los operadores de comparación:

$$>, >=, <, <=, ==, != $$

```
struct Racional {
    int num;
    int den;       // den > 0
};

int main() {
    Racional r;
    ...
}
```

```
struct Reloj {
    int hora;     // 0 <= hora < 24
    int min;      // 0 <= min < 60
    int sec;      // 0 <= sec < 60
};

int main() {
    Reloj r;
    ...
}
```

```
struct Film {
    string title;
    int year;
};

int main() {
    Film peli;
    ...
}
```

Sintaxis:

```
nombre_var.nombre_campo
```

Semántica:

- Se accede al campo *nombre_campo* de la variable *nombre_var*.
- Para que sea correcto, el tipo de datos de la variable ha de tener un campo llamado así.
- Este acceso es válido tanto para consultar el valor de ese campo como para modificarlo.

```
...
using namespace std;

struct Racional {
    int num;
    int den;      // den > 0
};

int main() {
    Racional r;
    cin >> r.num >> r.den;
    ....
    cout << r.num/double(r.den) << endl;
}
```

```
...
using namespace std;

struct Reloj {
    int hora;     // 0 <= hora < 24
    int min;      // 0 <= min < 60
    int sec;      // 0 <= sec < 60
};

int main() {
    Reloj r;
    cin >> r.hora >> r.min >> r.sec;
    ++r.sec;
    ...
    cout << r.hora << ':' << r.min << ...;
}
```

Sintaxis:

```
Tupla_A t;
// inicialización de los campos de t
...
// asignación a otra variable
Tupla_A copia = t;
```

Semántica:

Se hace una asignación campo a campo de la variable *t* a la variable *copia*.

```
int main() {
    Racional r;
    cin >> r.num >> r.den;
    Racional copia = r;          // lo que hace:   copia.num = r.num; copia.den = r.den;
    copia.den = copia.den + 5;
    cout << r.den << ' ' << copia.den << endl;
}
```

¡Alerta!

Depende del tipo de datos de los campos, esta asignación puede ser muy costosa! Si es así, pensar si es absolutamente necesario hacerla.

Paso de parámetros

Por valor:

- Recuerda que el paso por valor se hace sobre los parámetros de entrada
- En la llamada, se hace copia de esos valores.
- Por tanto, tenemos que pensar:
  - Paso por valor estándar: si la tupla tiene pocos valores (copia no costosa)

    ```
    void mi_accion(Racional r) {
        ...
    }
    ```

  - Referencia constante: si la tupla tiene muchos valores (copia costosa)

    ```
    void mi_accion(const Persona& p) {    // tupla definida más adelante
        ...
    }
    ```

Por referencia:

- Recuerda que el paso por referencia se hace sobre los parámetros de salida o de entrada/salida.
- Nada nuevo

Paso de parámetros: paso por valor estándar

Escribe una función tal que, dados dos racionales a y b (ambos con denominadores positivos), retorne cierto si a es mayor que b, false en caso contrario.

```
// Pre: a.den > 0, b.den > 0
// Post: retorna true si a mayor que b, false en caso contrario
bool mayor(Racional a, Racional b) {
    return a.num*b.den > b.num*a.den;
}
```

Escribe una función tal que, dados dos racionales a y b (ambos con denominadores positivos), retorne cierto si a es igual que b, false en caso contrario.

```
// Pre: a.den > 0, b.den > 0
// Post: retorna true si a igual que b, false en caso contrario
bool igual(Racional a, Racional b) {
    return a.num*b.den == b.num*a.den;
}
```

Paso de parámetros: paso por referencia

Escribe un procedimiento tal que, dado un reloj válido r, le sume un segundo.

```
// Pre: r es válido
// Post: suma un segundo al reloj y lo deja con el formato correcto
void suma_segundo(Reloj& r) {
    ++r.sec;
    if (r.sec == 60) {
        r.sec = 0;
        ++r.min;
        if (r.min == 60) {
            r.min = 0;
            ++r.hora;
            if (r.hora == 24) r.hora = 0;
        }
    }
}

// Pre: en la entrada hay 3 enteros que representan hora, minutos y segundos
// Post: escribir la hora, minutos y segundos de la entrada incrementada en 1 segundo
int main() {
    Reloj mi_r;     // mejor llamarlo r (objetivo: incidir en el paso por referencia)
    cin >> mi_r.hora >> mi_r.min >> mi_r.sec;
    suma_segundo(mi_r);
    cout << mi_r.hora << ' ' << mi_r.min << ' ' << mi_r.sec << endl;
}
```

Retorno de una función

Escribe una función tal que retorne un reloj inicializado a media noche.

```
// Pre: —
// Post: retorna un reloj inicializado a media noche
Reloj medianoche() {
    Reloj r;
    r.hora = 0;
    r.min = 0;
    r.sec = 0;
    return r;
}

// Uso de la función
int main() {
    ...
    Reloj mi_r = medianoche();
    ...
}
```

Tuplas anidadas

Nif { dni (int), letra (char) }

Persona { nombre (string), nif (Nif), edad (int) }

```
...
using namespace std;

struct Nif {
    int dni;        // dni > 0
    char letra;
};

struct Persona {
    string nombre;
    Nif nif;
    int edad;       // edad > 0
};

int main() {
    Persona p;
    p.nombre = "Arnau";  // cin >> p.nombre;
    p.edad = 14;
    p.nif.dni = 45789;     // p.nif es de tipo Nif (tiene dos campos: dni, letra)
    p.nif.letra = 'E';
    ...
    p.edad = p.edad + 6;
    cout << p.nif.dni << " " << p.edad;
}
```

Vectores de tuplas (búsqueda)

Escribe una función tal que, dado un vector de racionales v y un racional r, retorne true si existe en v algún racional que represente el mismo valor que r, false en caso contrario.

```
// Pre: v es válido
// Post: retorna true si r está en v, false en caso contrario
bool esta(const vector<Racional>& v, Racional r) {
    int n = v.size();
    for (int i = 0; i < n; ++i) {
        if (igual(v[i], r)) return true;
    }
    return false;
}
```

Vectores de tuplas (ordenación)

Queremos ordenar un vector de personas por los siguientes criterios:

(1) lexicográficamente creciente por nombre;

(2) para las personas con un mismo nombre, decreciente respecto la edad;

(3) para las que también tengan la misma edad, creciente según el dni.

```
// Pre: —
// Post: retorna true si a cumple los criterios de ordenación sobre b,
//               false en caso contrario
bool comp(const Persona& a, const Persona& b) {
    if (a.nombre != b.nombre) return a.nombre < b.nombre;
    if (a.edad != b.edad) return a.edad > b.edad;
    return a.nif.dni < b.nif.dni;
}

int main() {
    int n;
    cin >> n;
    vector<Persona> per(n);
    ...
    sort(per.begin(), per.end(), comp);
    ...
}
```

Otros ejemplos: Palabra más frecuente (algoritmo 3)

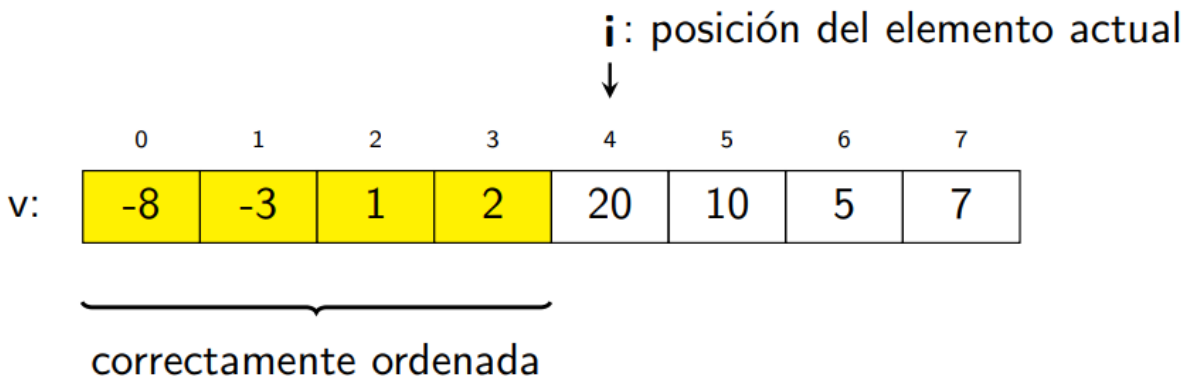
```
// Pre: 0 <= i < seq.size()
// Post: retorna el número de apariciones en seq de la palabra seq[i] desde la posición i
int cuantas_apariciones(const vector<string>& seq, int i) {
    int cont = 1;
    int j = i + 1;
    while (j < seq.size() and seq[i] == seq[j]) {
        ++cont;
        ++j;
    }
    return cont;
}

int main() {
    int n;
    cin >> n;
    while (n > 0) {
        vector<string> seq(n);
        for (int i = 0; i < n; ++i) cin >> seq[i];
        sort(seq.begin(), seq.end());  // seq ordenada ascendente lexicográficamente
        int i = 0;
        int max = 0;
        string word = "";
        while (i < n) {
            int repes = cuantas_apariciones(seq, i);
            if (repes >= max) {  // se incluye igualdad porque seq ordenada asc.
                word = seq[i];
                max = repes;
            }
            i = i + repes;
        }
        cout << word << endl;
        cin >> n;
    }
```

Otros ejemplos: Suma de vectores comprimidos

```
// Post: retorna un vector comprimido que representa la suma de v1 y v2
Vec_Com suma(const Vec_Com& v1, const Vec_Com& v2) {                    // sin push_back
    Vec_Com suma(v1.size() + v2.size());
    int i = 0;
    int j = 0;
    int k = 0;
    while (i < v1.size() and j < v2.size()) {
        if (v1[i].pos < v2[j].pos) {
            suma[k] = v1[i];
            ++i;
            ++k;
        } else if (v1[i].pos > v2[j].pos) {
            suma[k] = v2[j];
            ++j;
            ++k;
        } else {
            int v = v1[i].valor + v2[j].valor;
            if (v != 0) {
                suma[k].valor = v;
                suma[k].pos = v1[i].pos;
                ++k;
            }
            ++i;
            ++j;
        }
    }
    // acabar de poner en suma el resto de contenido de v1 o v2
    ...
    // crear el vector a devolver con exactamente k casillas
    Vec_Com res(k);
    for (int p = 0; p < k; ++p) res[p] = suma[p];
    return res;
}
```

Ordenación por selección (vector<int>)



Características:

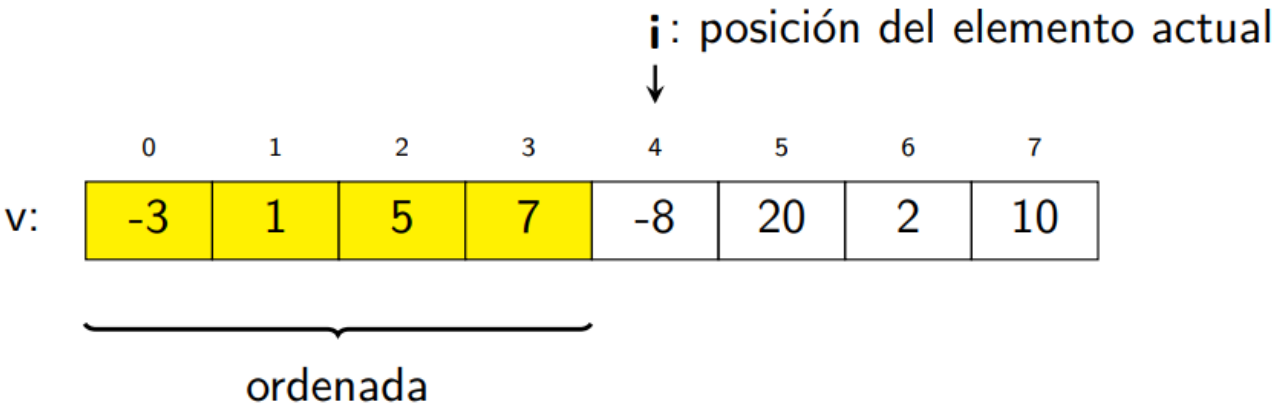
- Parte tratada: correctamente ordenada según operador <
- Tratamiento elemento actual: Intercambiar v[i] con el elemento menor de v[i, ..., v.size() - 1].

```
// 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 inserción (vector<int>)



Características:

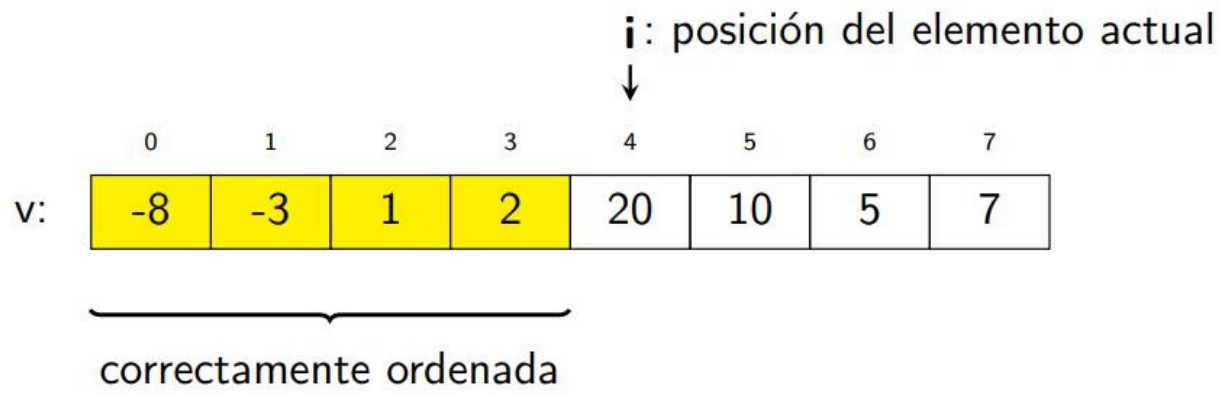
- Parte tratada: ordenada según operador $<$
- Tratamiento elemento actual: Llevar elemento v[i] a su posición correcta en v[0, ..., i], que quedará ordenado.

```
// 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;    // posición en la que compruebo si tiene que ir x
        while (j > 0 and v[j - 1] > x) {
            v[j] = v[j - 1];
            --j;
        }
        // al salir del bucle, j es la posición que buscábamos
        v[j] = x;
    }
}
```

```
// 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(x, 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: Levar a la posición i el elemento menor de v[i, ..., v.size() - 1] comparando dos a dos los elementos desde el final del vector hasta la posición i.

Ordenación de la burbuja (vector<int>)

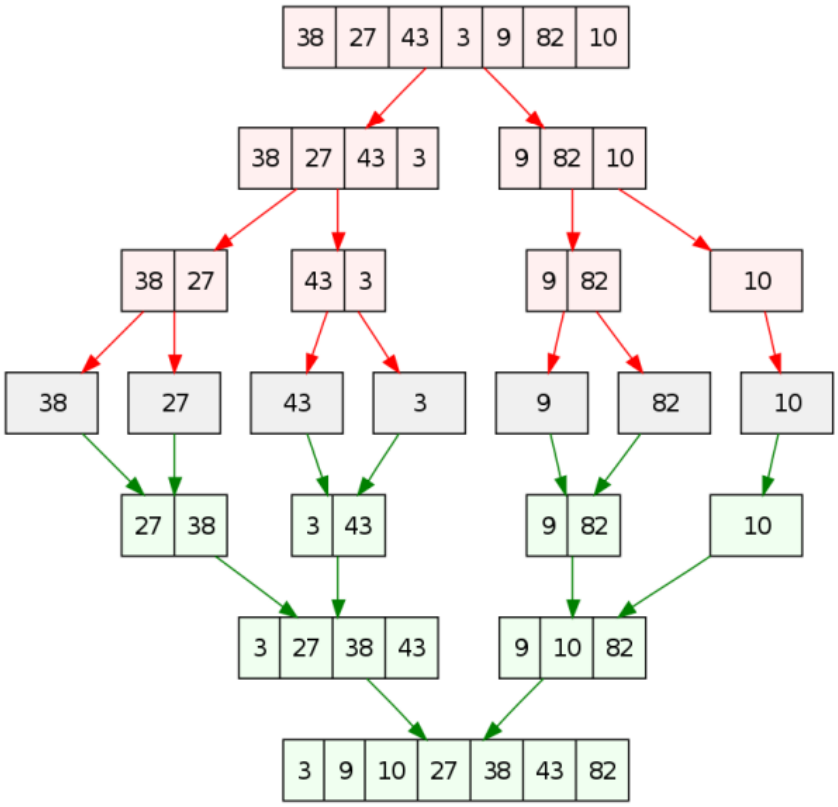
```
// Pre: a vale A, b vale B
// Post: a vale B, b vale A
void intercambiar(int& a, int& b) {
    int aux = a;
    a = b;
    b = aux;
}

// 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 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);
```