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: 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
}
```

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();
    // ...
}
```

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

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;
    }
```

```
...
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 << ...;
}
```

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();
    ...
}
```

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);
    ...
}
```

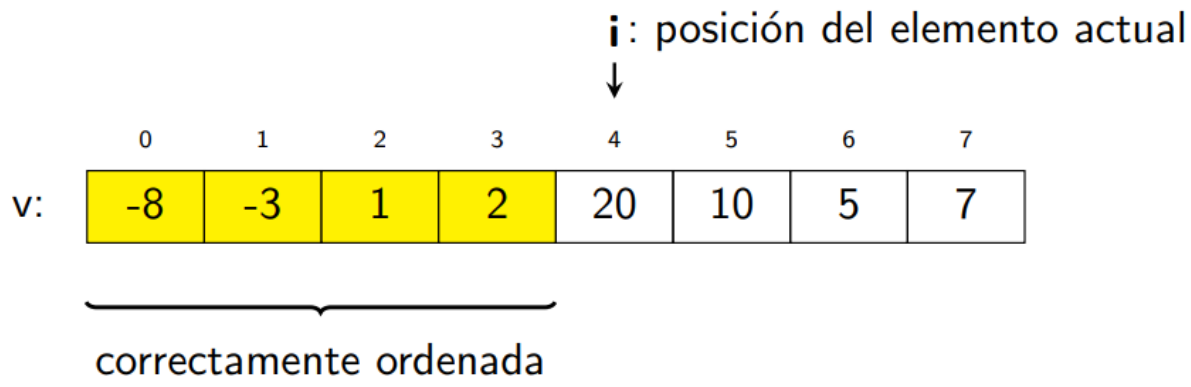
Otros ejemplos: Palabra más frecuente (algoritmo 3)

```
// 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:

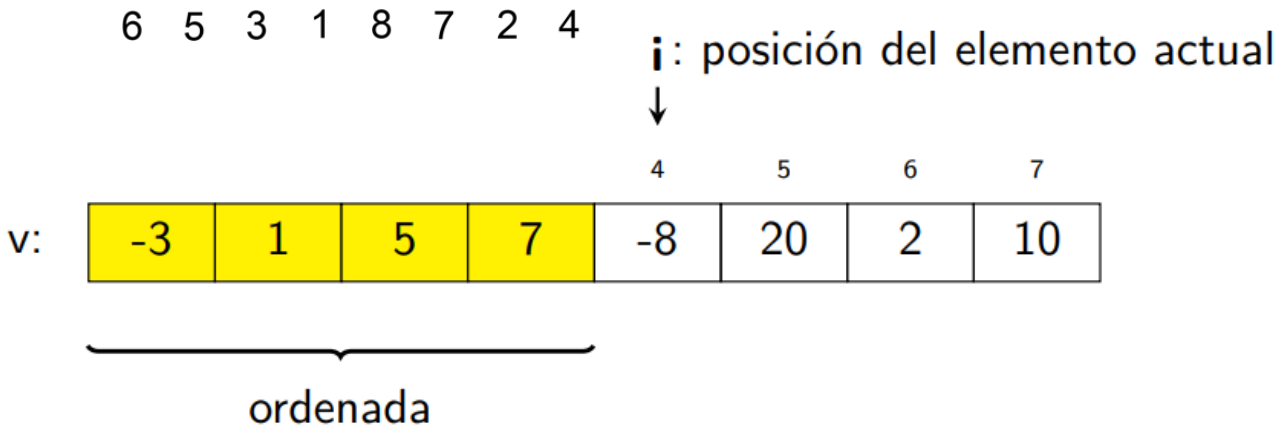
- Parte tratada: correctamente ordenada según operador <
- Tratamiento elemento actual: Intercambiar v[i] con el elemento menor de v[i, ..., v.size() - 1].

```
// 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]);
    }
}
```

8
5
2
6
9
3
1
4
0
7

https://drive.google.com/file/d/1l5elat0B3GmlQKDVyM6AANnf-ZV6wBcY/view?usp=share_link

Ordenación por inserción (vector<int>)



Características:

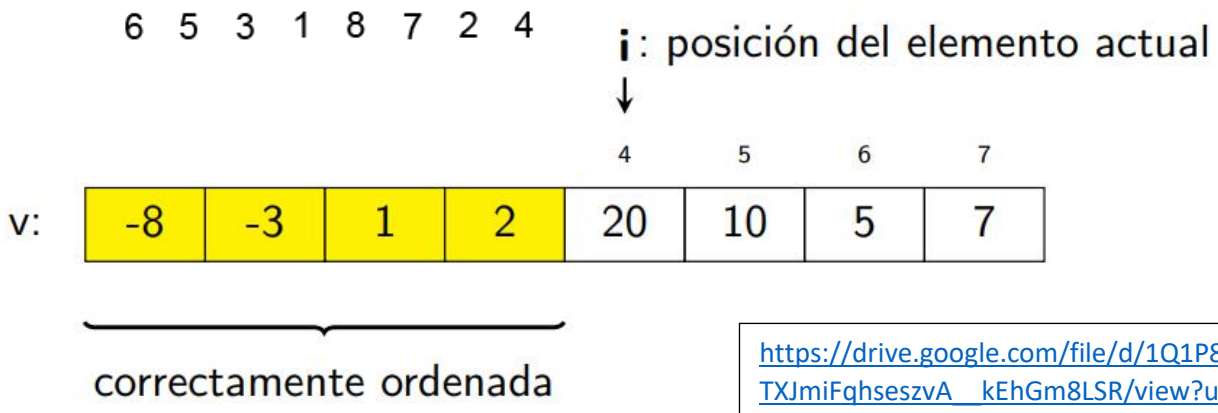
- Parte tratada: ordenada según operador <
- Tratamiento elemento actual: Llevar elemento v[i] a su posición correcta en v[0, ..., i], que quedará ordenado.

```
// 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;
    }
}
```

https://drive.google.com/file/d/1d-5ZVkBgF9FwLxFzh6LbEptZMcG2cMIO/view?usp=share_link

Ordenación de la burbuja (vector<int>)



https://drive.google.com/file/d/1Q1P83J-TXJmiFqhseszvA__kEhGm8LSR/view?usp=share_link

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.

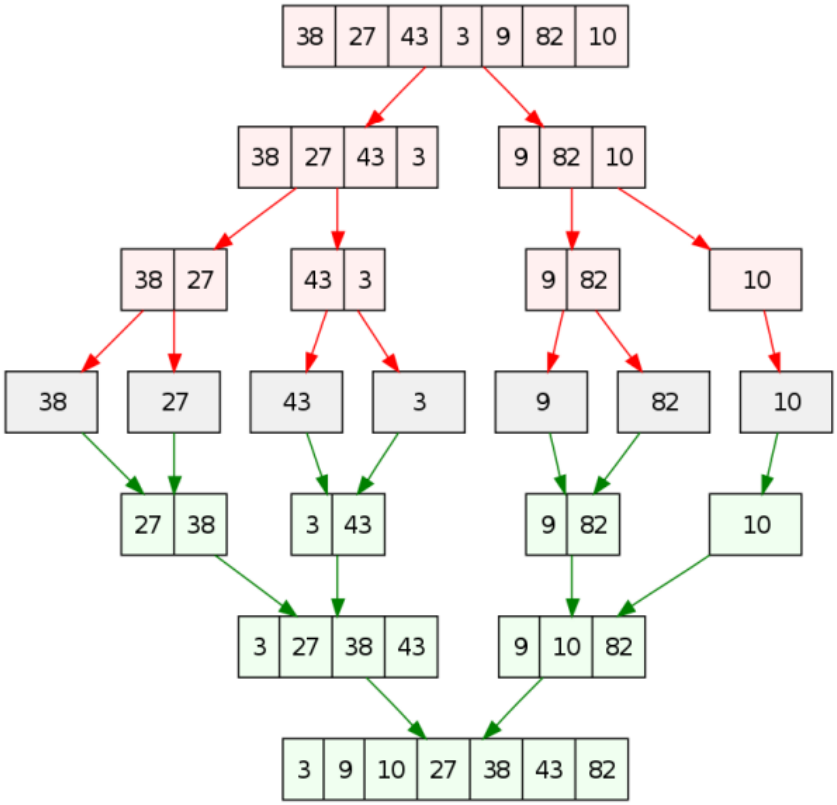
Ordenación de la burbuja (vector<int>)

```
// 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>)

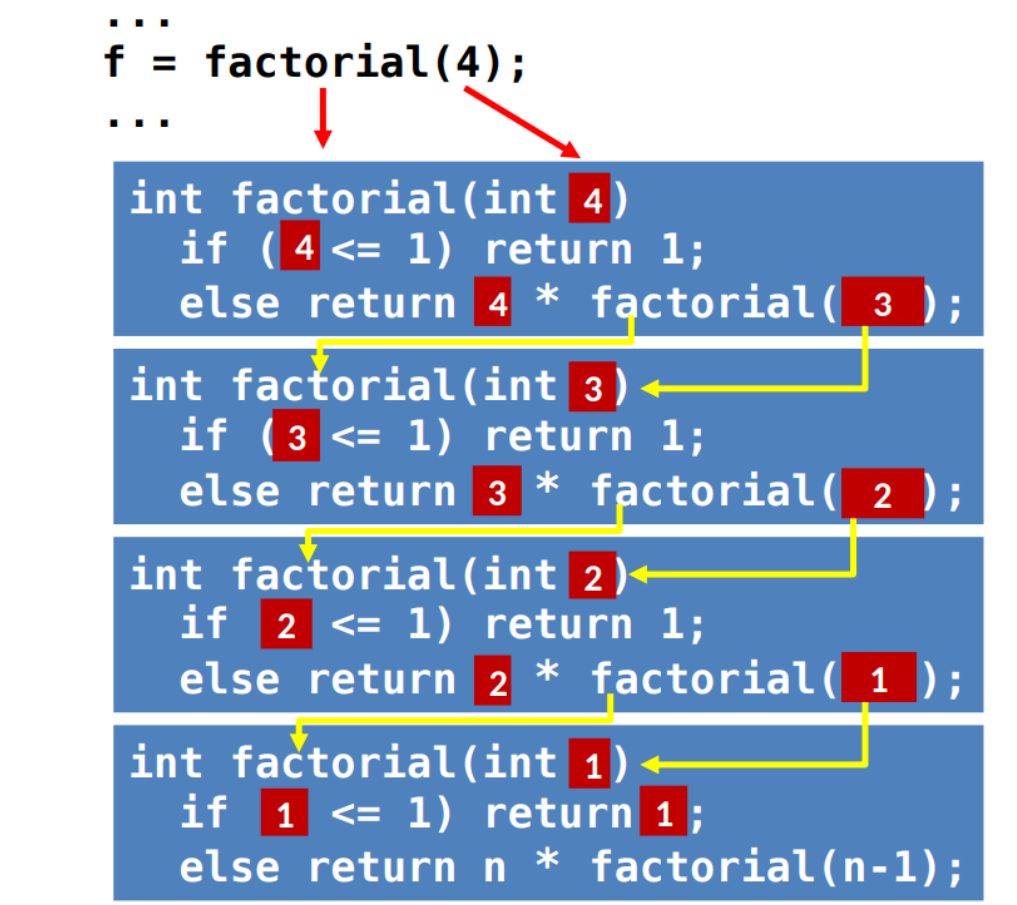


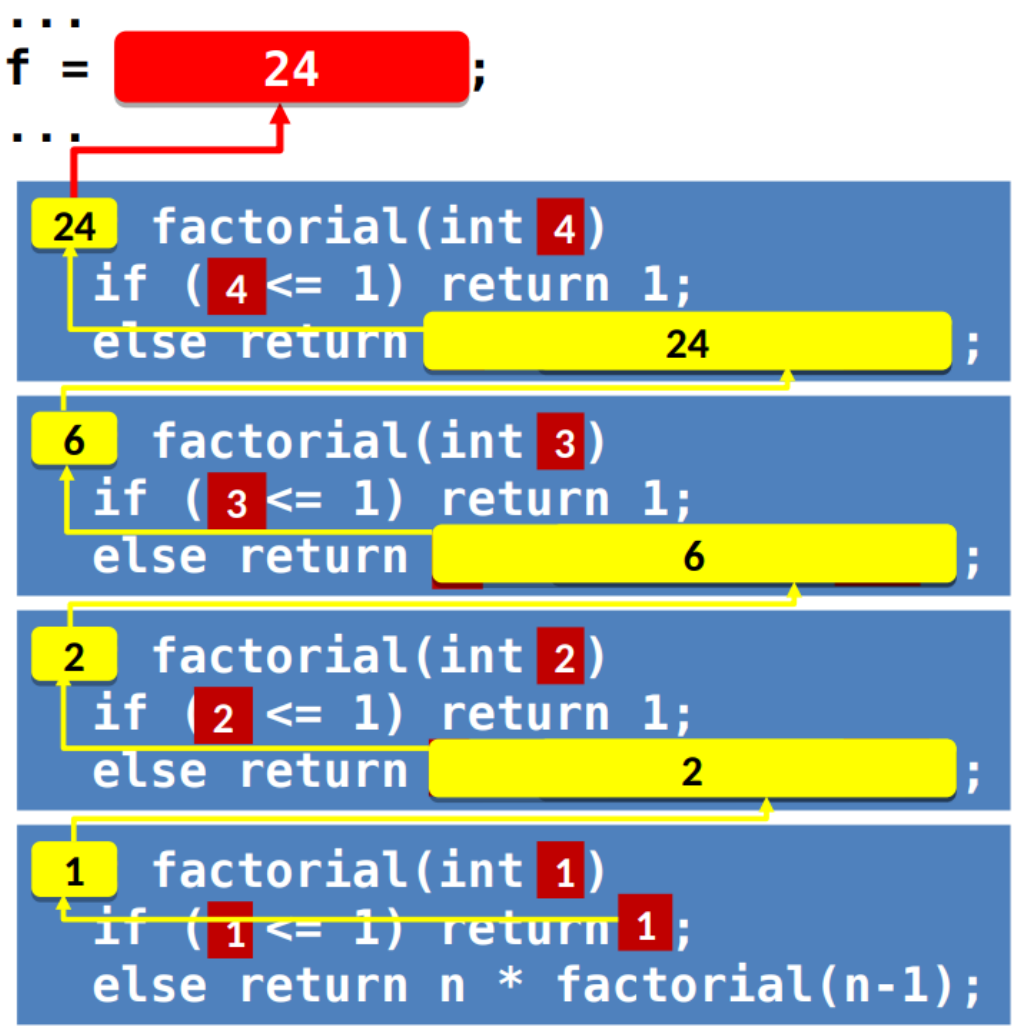

```
// Pre: v válido, 0 <= esq <= m < dre < v.size()
// Post: v[esq ... dre] queda ordenado según operador <
void fusion(vector<int>& v, int esq, int m, int dre) {
    int n = dre - esq + 1;
    vector<int> aux(n);
    int k = 0;  // índice sobre aux
    int i = esq;
    int j = m + 1;
    while (i 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

6 5 3 1 8 7 2 4

https://drive.google.com/file/d/1RXiEv9bwlk85QS_2fJEsL6Jdf5VkXfWb/view?usp=share_link

Recursion: behind the scenes

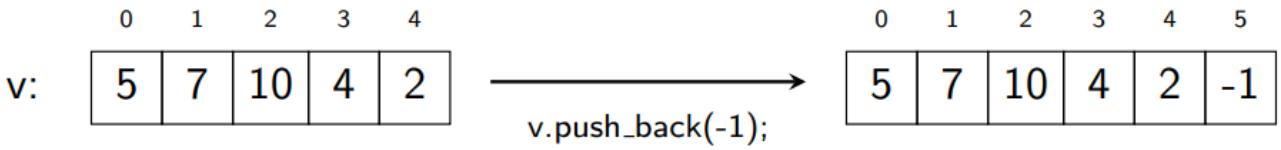




Añadir un nuevo elemento:

```
nombre_vector.push_back(valor_T);
```

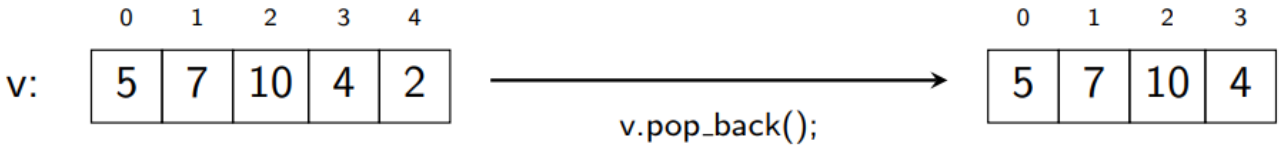
- Al vector *nombre_vector* se le añade un nuevo último elemento con valor *valor_T* y tipo de datos T (el mismo que el del vector).



Borrar último elemento del vector:

```
nombre_vector.pop_back();
```

- Al vector *nombre_vector* se le quita su último elemento.



Typedef

Sintaxis:

```
typedef tipo_de_datos_existente nuevo_nombre;
```

Semántica:

- *nuevo_nombre* es sinónimo de *tipo_de_datos_existente*.

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

typedef vector<int> Polinomio;

int evaluar_polinomio(const Polinomio& p, int x) {
    ...
}

int main() {
    int x, n;
    cin >> x >> n;
    Polinomio p(n);
    for (int i = 0; i < n; ++i) cin >> p[i];
    cout << evaluar_polinomio(p, x) << endl;
}
```

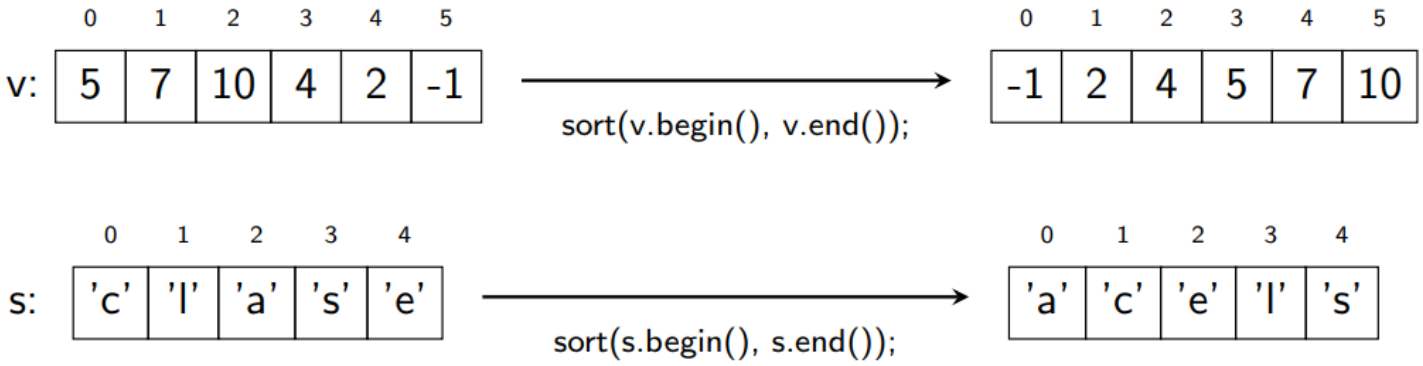
Ordenación (biblioteca algorithm)

Sintaxis:

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

sort(nombre_vector.begin(), nombre_vector.end());
```

Semántica: Se ordena todo el vector *nombre_vector*, en orden ascendente según el operador $<$ definido para el tipo de datos de ese vector.

Ejemplo:



Ordenación (biblioteca algorithm)

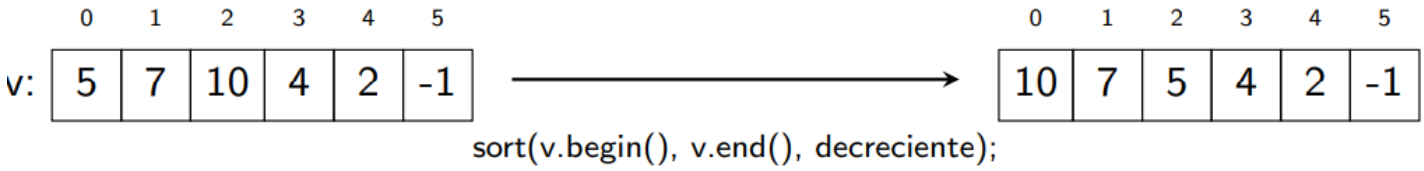
Sintaxis:

```
// 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) {...}

sort(nom_v.begin(), nom_v.end(), nom_func_bool);
```

Semántica:
Se ordena todo el vector *nombre_v*, en orden ascendente según la función booleana *nom_func_bool*. T es el tipo de datos del vector. El orden es estricto (es decir, *nom_func_bool(a, a)*$\rightarrow$ false).

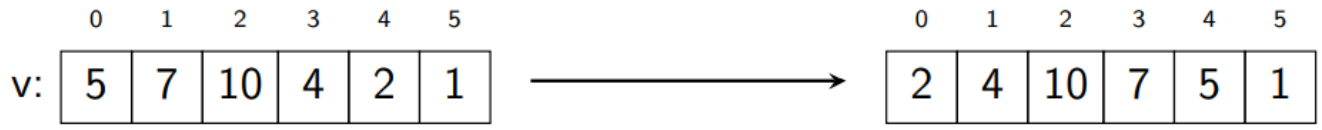
Ejemplo:

```
// quiero que a vaya antes que b en el vector cuando a > b
bool decreciente(int a, int b) {
    if (a > b) return true;
    else return false;
}
```



Escribir un procedimiento tal que dado un vector de enteros, lo transforme de tal manera que sus elementos pares aparezcan antes que los elementos impares. Además, los elementos pares estarán ordenados de forma ascendente, mientras que los impares de forma descendente.

Ejemplo:




```
bool orden(int a, int b) {
    bool par_a = a%2 == 0;
    bool par_b = b%2 == 0;

    if (par_a != par_b) return par_a;      // tienen diferente paridad
    if (par_a) return a < b;               // los dos son pares
    return a > b;                          // los dos son impares
}

void separar(vector <int>& v) {
    sort(v.begin(), v.end(), orden);
}
```

Recursividad aplicada: Máximo Común Divisor

Ooootro post sobre **recursividad**, pero no se preocupen ya solo queda este y otro. Esta vez vamos a calcular el **máximo común divisor** de dos números de **forma recursiva en c++**.

```
#include<stdio.h>
int MCD(int x, int y)
{
    if(y==0)
        return x;
    else
        return MCD(y, x%y);
}
```

Ordenar por filas

```
typedef vector<int> Row;
typedef vector<Row> Matrix;

void read_matrix(Matrix& mat) {
    int m = mat.size();
    int n = mat[0].size();
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> mat[i][j] ;
        }
    }
}

void write_matrix(const Matrix& mat) {
    int m = mat.size();
    int n = mat[0].size();
    for (int i = 0; i < m; ++i) {
        cout << mat[i][0];
        for (int j = 1; j < n; ++j) cout << ' ' << mat[i][j];
        cout << endl;
    }
}
```

```
// Pre: u, v are two rows of 0 and 1 values; each represents a binary number;
//         u.size() = v.size()
// Post: returns true when (number) u is greater than (number) v,
//       returns false otherwise
bool comp(const Row& u, const Row& v) {
    int n = u.size();
    if (u[0] != v[0]) return v[0] == 1;
    //here, u[0] == v[0]
    int i = 1;
    while (i < n and u[i] == v[i]) ++i;
    if (u[0] == 1) return i < n and u[i] == 0;
    else return i < n and u[i] == 1;
}

bool comp(const Row& u, const Row& v) {  // alternativa
    int n = u.size();
    if (u[0] != v[0]) return v[0] == 1;
    //here, u[0] == v[0]
    for (int i = 1; i < n; ++i) {
        if (u[i] != v[i]) {
            if (u[0] == 1) return u[i] == 0;
            else return u[i] == 1;
        }
    }
    return false;
}

int main() {
    int m, n;
    while (cin >> m >> n) {
        Matrix mat(m, Row(n));
        read_matrix(mat);
        sort(mat.begin(), mat.end(), comp);
        write_matrix(mat);
        cout << endl;
    }
}
```