```cpp
#include <bits/stdc++.h>
using namespace std;
#define pb push_back
#define all(a) (a).begin(), (a).end()
using ll = long long;
void solve(){

}

int main(){
    std::cin.tie(nullptr);
    std::ios_base::sync_with_stdio(false);
    int t = 0; cin >> t;
    while(t--){solve();}
    return 0;
}
```

```cpp
std::cin.tie(nullptr);
std::ios_base::sync_with_stdio(false);
```

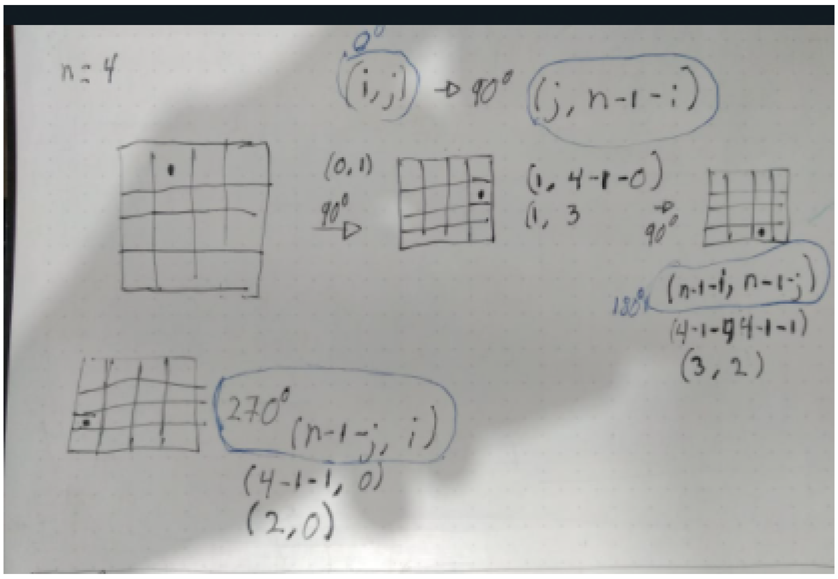
int dif = 1e9; = 10000000001

10^8 operations can be done in 1 sec

☐ Algorithms

Matrix rotation

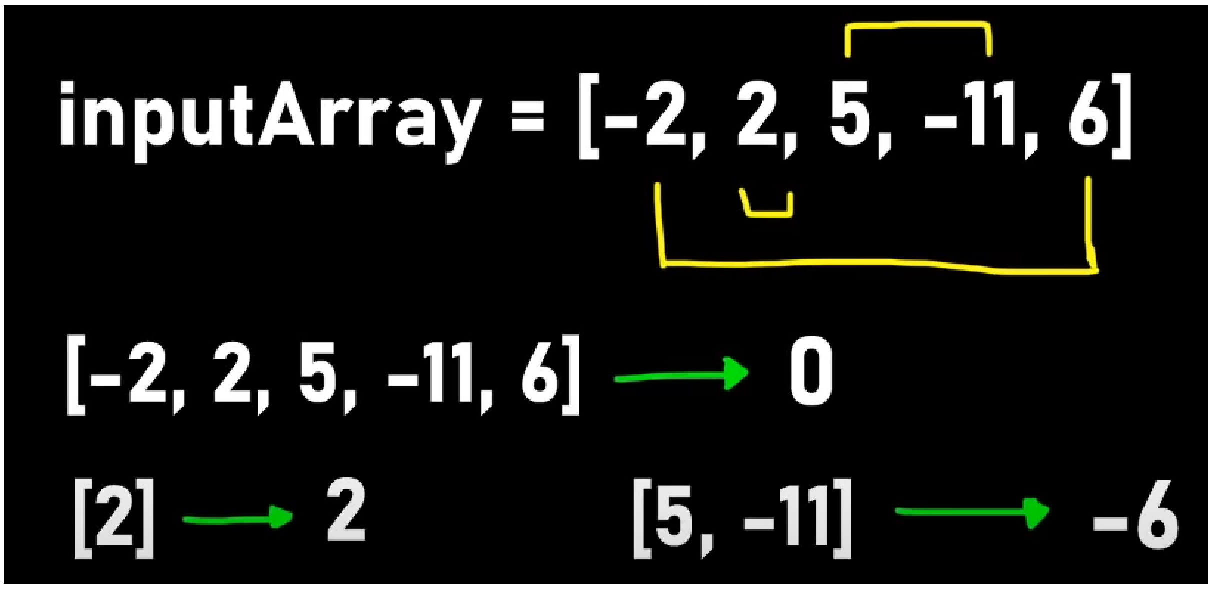
```
n = size of matrix
for (i < n)
        for(j < n)
m[i][j] = pos actual
m[j][n - 1 - i] = 90 degrees
m[n - 1 - i][n - 1 - j] = 180 degrees rotation
m[n - 1 - j][i] = 270 degrees
```



Kadane´s algorithm

Maximum sum subarray

We are looking for the maximum sum we can get from a contiguous subarray.

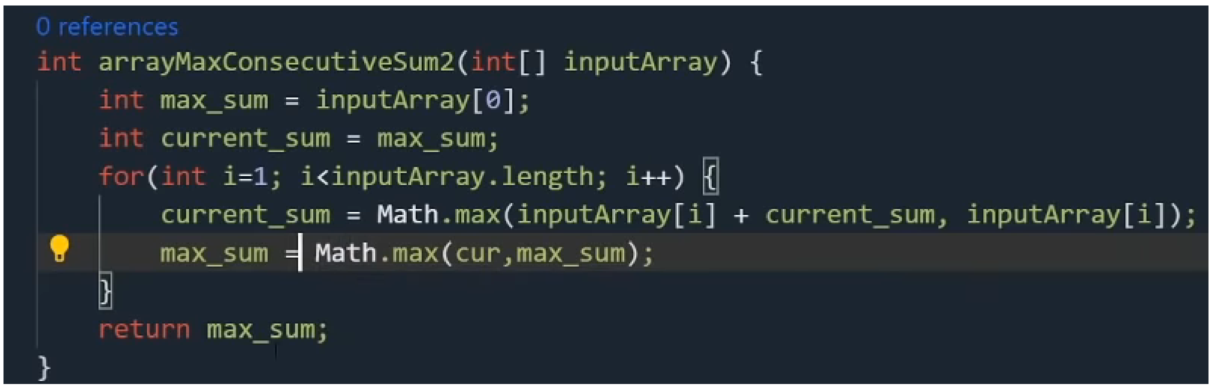


inputArray = [-2, 2, 5, -11, 6]

[-2, 2, 5, -11, 6] → 0

[2] → 2

[5, -11] → -6



```
0 references
int arrayMaxConsecutiveSum2(int[] inputArray) {
    int max_sum = inputArray[0];
    int current_sum = max_sum;
    for(int i=1; i<inputArray.length; i++) {
        current_sum = Math.max(inputArray[i] + current_sum, inputArray[i]);
        max_sum = Math.max(cur,max_sum);
    }
    return max_sum;
}
```

Dynamic Programming

Coin Change

¿De cuántas maneras puedes pagar cierta cantidad con un arreglo limitado de monedas?
Tenemos nuestro arreglo de monedas.

Para pagar 1 peso solo tenemos una opción que sería con la moneda de $1.
Si suponemos que solo hay monedas de 1 $

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Para ir calculando cuando de nuevo los valores al agregar una moneda nueva lo que hacemos es:

Estamos sumando lo que tenemos en nuestra casilla + lo que está en 2 posiciones anteriores.
X[j] += X[j - k];
A nuestra posición le sumamos j(pos) - k(actualCoin)

Coin Change

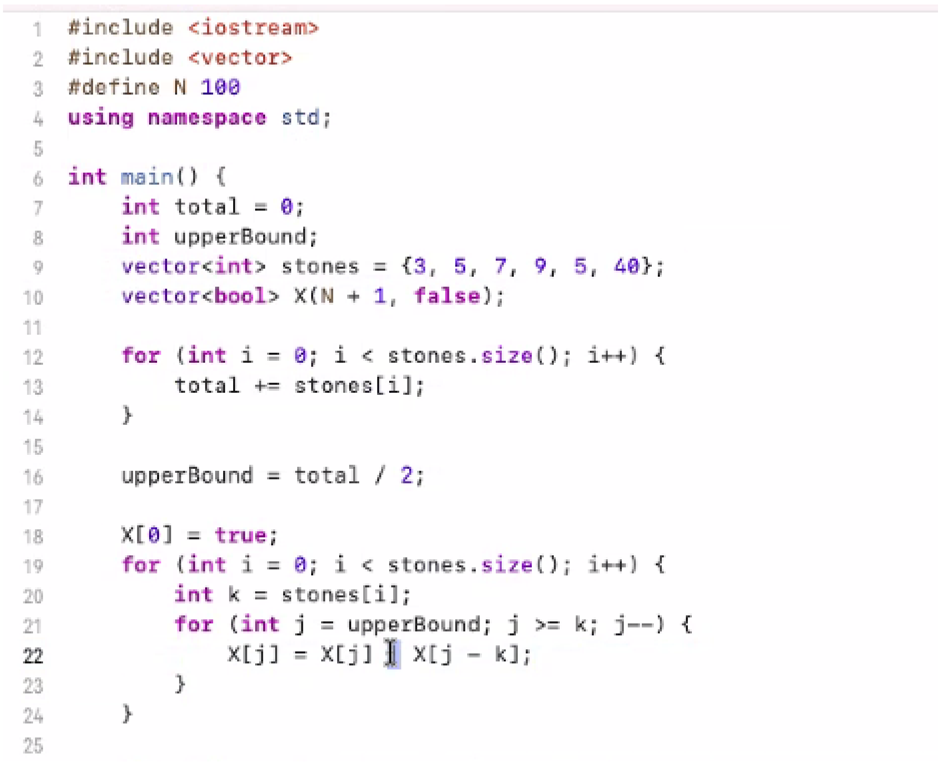
Mínimo número de monedas para pagar cierta cantidad.
Tenemos nuestras monedas, y la cantidad a pagar.

Torre de Piedras

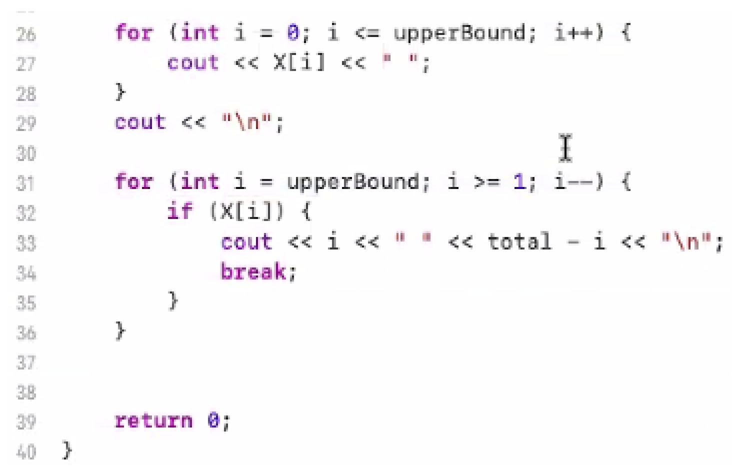
O(nm)
N = cantidad de piedras. M = Resultado de dividir entre 2 el peso total.
Sumamos el peso total de las piedras. Dividimos entre 2. Buscamos las maneras de sumar ese número el tamaño de las piedras que tenemos.
Arreglo de bool X por que solo nos interesa saber si se puede hacer cierta cantidad.
Checamos cuales cantidades podemos hacer con nuestras piedras.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	0	1	1	0	1	1	1	0	1	1	0	0	1

El 14 lo podemos sumar con la de 8+3+3
Y en el otro montón pongo lo restante.


```cpp
#include <iostream>
#include <vector>
#define N 100
using namespace std;

int main() {
    int total = 0;
    int upperBound;
    vector<int> stones = {3, 5, 7, 9, 5, 40};
    vector<bool> X(N + 1, false);

    for (int i = 0; i < stones.size(); i++) {
        total += stones[i];
    }

    upperBound = total / 2;

    X[0] = true;
    for (int i = 0; i < stones.size(); i++) {
        int k = stones[i];
        for (int j = upperBound; j >= k; j--) {
            X[j] = X[j] | X[j - k];
        }
    }

```

K es nuestra piedra actual.
2do for:
j = upperBound mientras que j >= k


```cpp
    for (int i = 0; i <= upperBound; i++) {
        cout << X[i] << " ";
    }
    cout << "\n";

    for (int i = upperBound; i >= 1; i--) {
        if (X[i]) {
            cout << i << " " << total - i << "\n";
            break;
        }
    }


    return 0;
}
```

Optimal Matrix Chain Multiplicación

Code:
#include <cstdio>
#include <cstring>
#define N 100
#define oo 1000000
using namespace std;

int A[N + 1];

```
int X[N + 1][N + 1];
int path[N +1 ][N + 1];

// X[i][j] es minimo numero de operaciones para multiplicar la seccion de i a j.
// Resultado X[0][n-1]

void printSequence(int, int);

int main() {
    int n, val;

    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &A[i], &A[i + 1]);
    }

    memset(X, 0, sizeof(X));
    for (int len = 1; len <= n - 1; len++) {
        for (int i = 1; i <= n - len; i++) {
            int j = i + len;
            X[i][j] = oo;
            for (int k = i; k <= j; k++) {
                val = X[i][k] + X[k + 1][j] + (A[i - 1] * A[j] * A[k]);
                if (val < X[i][j]) {
                    X[i][j] = val;
                    path[i][j] = k;
                }
            }
        }
    }

    printf("%d\n", X[1][n]);
    printSequence(1, n);
    printf("\n");

    return 0;
}

void printSequence(int a, int b) {
    if (a == b) {
        printf("A%d", a);
    } else {
        printf("(");
        printSequence(a, path[a][b]);
        printf("x");
        printSequence(path[a][b] + 1, b);
        printf(")");
    }
```

}

KMP

Pattern searching in a string. String searching.
O(n)
We use the pref func to know which is the Largest Palindromic String.

```
vector<int> pref(string pattern){
    vector<int> lps(pattern.size());
    int idx = 0;
    for(int i = 1; i < pattern.size();){
        if(pattern[i] == pattern[idx]){
            lps[i] = idx + 1;
            i++;
            idx;
        }else{
            if(idx != 0){
                idx = lps[idx - 1];
            }else{
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}
int KMP(string t, string pattern){
    vector<int> lps = pref(pattern);
    int i = 0, j = 0;
    while(i < t.size() && j < pattern.size()){
        if(t[i] == pattern[j]){
            i++;
            j++;
        }else{
            if(j != 0) j = lps[j - 1];
            else i++;
        }
    }
    if(j == pattern.size()) return true;
    else return false;
}
```

Strings

substr(pos, len)
pos: Position of the first character to be copied as a substring.
len: Number of characters to include in the substring.

You can use substr to create prefixes and suffixes of string.
Example:

```
for(int i = 0; i < n; i++){
        bool ok = false;
        for(int j = 0; j < s[i].size(); j++){
            string pref = s[i].substr(0, j);
            string suff = s[i].substr(j, s[i].length() - j);//Incluyendo j agarra s[i.length - j]
            if(M[pref] && M[suff]) ok = true;
        }
        cout << ok;
    }
```

String Hashing

B, M Tienen que ser primos.
B tiene que ser mayor al valor de de nuestras letras.
En **powB** guardamos las potencias de B.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll M = pow(10, 9) + 7;
int B = 157;

int v(char c){
    return c -'a' + 1;
}

void solve(){
    string s, p;
    cin >> s >> p;
    int n = s.size(), m = p.size();

    vector<ll> powB(n + 1), H(n + 1), Ht(m + 1);
    H[0] = 0;
    Ht[0] = 0;
    powB[0] = 1;

    for(int i = 0; i < n; i++){//Calculate Hashes.
        H[i + 1] = (H[i] * B + v(s[i])) % M;
        powB[i + 1] = (powB[i] * B) % M;
    }

    for(int i = 0; i < m; i++){
        Ht[i + 1] = (Ht[i] * B + v(p[i])) % M;
        powB[i + 1] = (powB[i] * B) % M;
    }

    int ans = 0;

    for(int i = 0; i + m <= n; i++){
        int res = ((H[i + m] - H[i] * powB[m]) % M + M ) % M;
        if(Ht.back() == res){
            ans += 1;
        }
    }

    /*cout << "Hashes :\n";
    for(int i = 0; i < n; i++) cout << H[i] << " ";
    cout << "\n";
    for(int i = 0; i < m; i++) cout << Ht[i] << " ";*/


    cout << ans;
}
```

Data structures

STL

Vector

Sort

you to include <algorithm>
sort(your_array.begin(),your_array.end());
This will sort through your elements in O(nlogn)

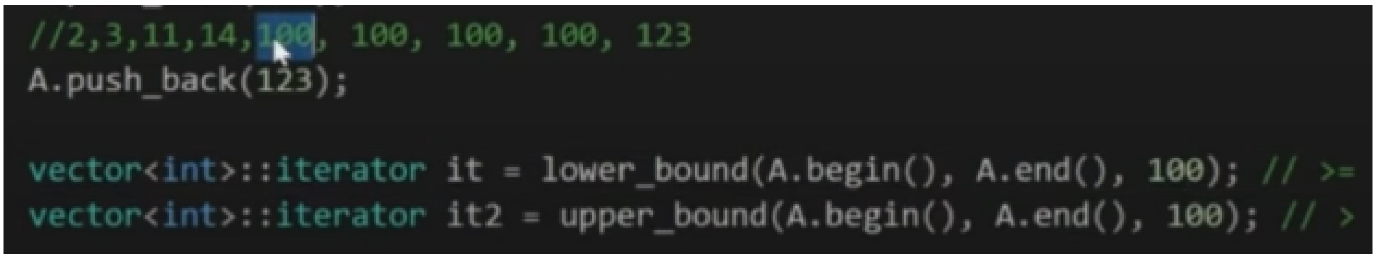
binary_search

To use binary search we need to have our array sorted.

binary_search(your_array.begin(),your_array.end(),variable_to_be_find);
It returns a boolean.
O(logn)

lower_bound()

Return an iterator pointing to the first element in the range [first, last) which has a value not less than val. This means that the function returns an iterator pointing to the next smallest number just greater than or equal to that number. If there are multiple values that are equal to val, lower_bound() returns the iterator of the first such value.



```
//2,3,11,14,100, 100, 100, 100, 123
A.push_back(123);

vector<int>::iterator it = lower_bound(A.begin(), A.end(), 100); // >=
vector<int>::iterator it2 = upper_bound(A.begin(), A.end(), 100); // >
```

You can use it to know how far is a number from a value.

```
for(int i = 1; i <= n; i++){
      if(N[i] >= i) continue;//Does not satisfy inequality.
      res += (long long)(lower_bound(v.begin(), v.end(), N[i]) -
v.begin());//Search how far is N[i] from the start of the vector v.
            v.push_back(i);//Make list storing all i that appeats
before j.
}
```

Set

set<data type> name;
Save unique elements. Values are stored in a specific order.

unordered_set<data type> name
insert()
erase()
Printing a set:
for (auto it = myset.begin(); it !=
                              myset.end(); ++it)
        cout << ' ' << *it;

Map

Each element has a key value and a mapped value. We use the key to access the mapped value.
`map<int, int> order`

```
insert(pair<int, int>(1, 40));
```

Erasing elements up to a certain value:

```
gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
```

Ways of printing a map:

Range-based for loops

```
for (auto const &pair: m) {
        std::cout << "{" << pair.first << ": " << pair.second << "}\n";
    }
```

Using an iterator

```
for (auto it = m.cbegin(); it != m.cend(); ++it) {
        std::cout << "{" << (*it).first << ": " << (*it).second << "}\n";
    }
```

Overloading <<

```
ostream &operator<<(std::ostream &os, const std::unordered_map<K, V> &m) {
    for (const std::pair<K, V> &p: m) {
        os << "{" << p.first << ": " << p.second << "}\n";
    }
    return os;
}
```

Traverse map in reverse direction

```
map<int, int>::reverse_iterator it;
for (it = mp.rbegin(); it != mp.rend(); it++) {
        cout << "(" << it->first << ", " << it->second << ")" << endl;
    }
```

Search if a key is in a map

```
if(mp.find(b) != mp.end()){} mp[b]++;
```

Trees

Traversals:

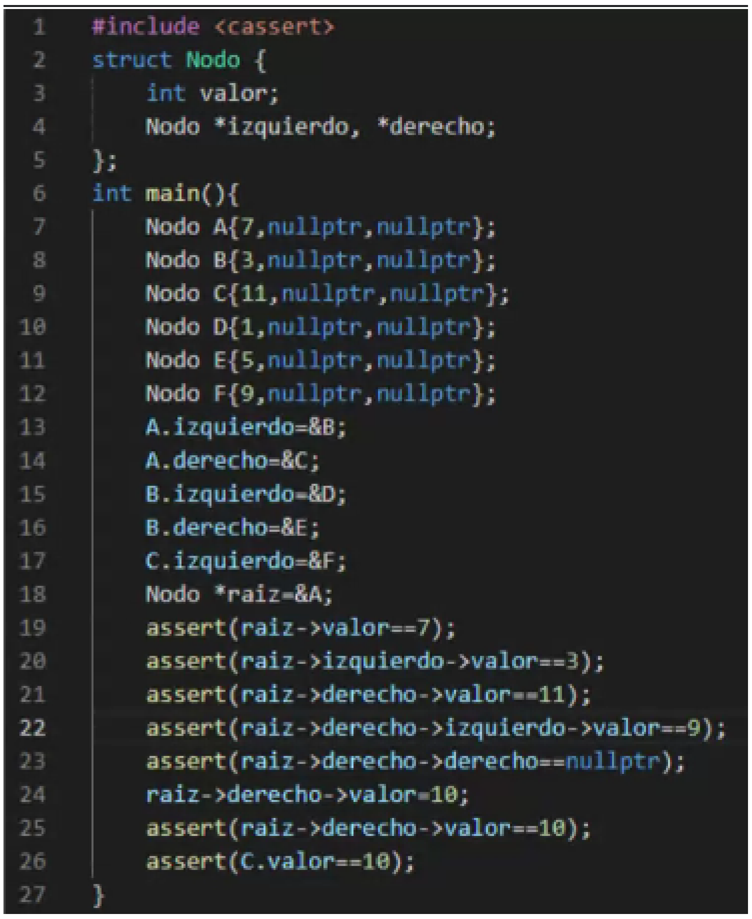
```
void preorder(Nodo* nodo){
            if(nodo == nullptr) return;
            cout << nodo->val << " ";
            preorder(nodo->left);
            preorder(nodo->right);
}

void order(Nodo* nodo){
            if(nodo == nullptr) return;
            order(nodo->left);
            cout << nodo->val << " ";
            order(nodo->right);
}

void postorder(Nodo* nodo){
            if(nodo == nullptr) return;
            postorder(nodo->left);
            postorder(nodo->right);
            cout << nodo->val << " ";
}
```

Levelorder

```
int altura(TreeNode* root){
        if(root == nullptr) return 0;
        int altura_izq = 1 + altura(root->left);
        int altura_der = 1 + altura(root->right);
        return max(altura_izq, altura_der);
    }

    void printCurrentLevel(TreeNode *root, int level){
        if(root == nullptr) return;

        if(level == 1) cout << root->val << " ";
        else if(level > 1){
            printCurrentLevel(root->left, level - 1);
            printCurrentLevel(root->right, level - 1);
        }
    }

    void levelTraversal(TreeNode* root){
        k = altura(root);
        for(int i = 1; i <= k; i++) printCurrentLevel(root, i);
    }
```

Binary Tree

Implementación:



```cpp
#include <cassert>
struct Nodo {
    int valor;
    Nodo *izquierdo, *derecho;
};
int main(){
    Nodo A{7,nullptr,nullptr};
    Nodo B{3,nullptr,nullptr};
    Nodo C{11,nullptr,nullptr};
    Nodo D{1,nullptr,nullptr};
    Nodo E{5,nullptr,nullptr};
    Nodo F{9,nullptr,nullptr};
    A.izquierdo=&B;
    A.derecho=&C;
    B.izquierdo=&D;
    B.derecho=&E;
    C.izquierdo=&F;
    Nodo *raiz=&A;
    assert(raiz->valor==7);
    assert(raiz->izquierdo->valor==3);
    assert(raiz->derecho->valor==11);
    assert(raiz->derecho->izquierdo->valor==9);
    assert(raiz->derecho->derecho==nullptr);
    raiz->derecho->valor=10;
    assert(raiz->derecho->valor==10);
    assert(C.valor==10);
}
```

Symmetric

```cpp
bool isMirror(TreeNode * t1, TreeNode* t2){
      if(t1 == nullptr && t2 == nullptr) return true;
      if(t1 == nullptr || t2 == nullptr) return false;

      return (t1->val == t2->val) &&
          isMirror(t1->left, t2->right) &&
          isMirror(t1->right, t2->left);
  }

bool isMirror(TreeNode * left, TreeNode* right){
       if(left == nullptr || right == nullptr) return left == right;

       if(left->val != right->val) return false;

       return isMirror(left->left, right->right) &&
           isMirror(left->right, right->left);
   }
```

PathSum

```
bool haspathSum(TreeNode* root, int targetSum, int pathSum){
        if(root == nullptr) return false;

        int a = targetSum;
        int sum = pathSum + root->val;
        if(root->left == nullptr && root->right == nullptr)
            return targetSum == sum;

        bool ans_left = haspathSum(root->left, a, sum);
        bool ans_right = haspathSum(root->right, a, sum);
        return ans_left || ans_right;
    }
```

Heap Tree

Easy and fast implementation can be done with a priority_queue.

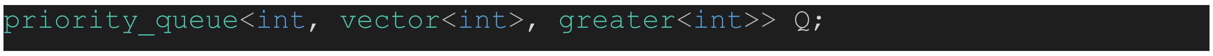
Priority_queue

A priority queue in c++ is a type of container adapter, which processes only the highest priority element, i.e. the first element will be the maximum of all elements in the queue, and elements are in decreasing order.

Declaration:
#include <queue>

priority_queue<int> Q;

For a priority queue in ascending order:



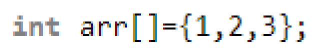
```
priority_queue<int, vector<int>, greater<int>> Q;
```

https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/

Note: The above syntax may be difficult to remember, so in case of numeric values, we can multiply the values with -1 and use max heap to get the effect of min heap.

We can multiply tha data that we want to store in the priority queue * -1

Then when we retrieve the data we can simply multiply by - 1. cout << pq.top()*-1;
Example:



```
int arr[]={1,2,3};
```

```
  for(int i=0; i<3; i++){
  arr[i]=-arr[i];             // multiplying each array element by -1
}

priority_queue<int> pq2(arr, arr+3);

  cout<<"Min priority queue: ";
  while(pq2.empty()==false){
  cout<<-pq2.top()<<" ";
  pq2.pop();
}
```

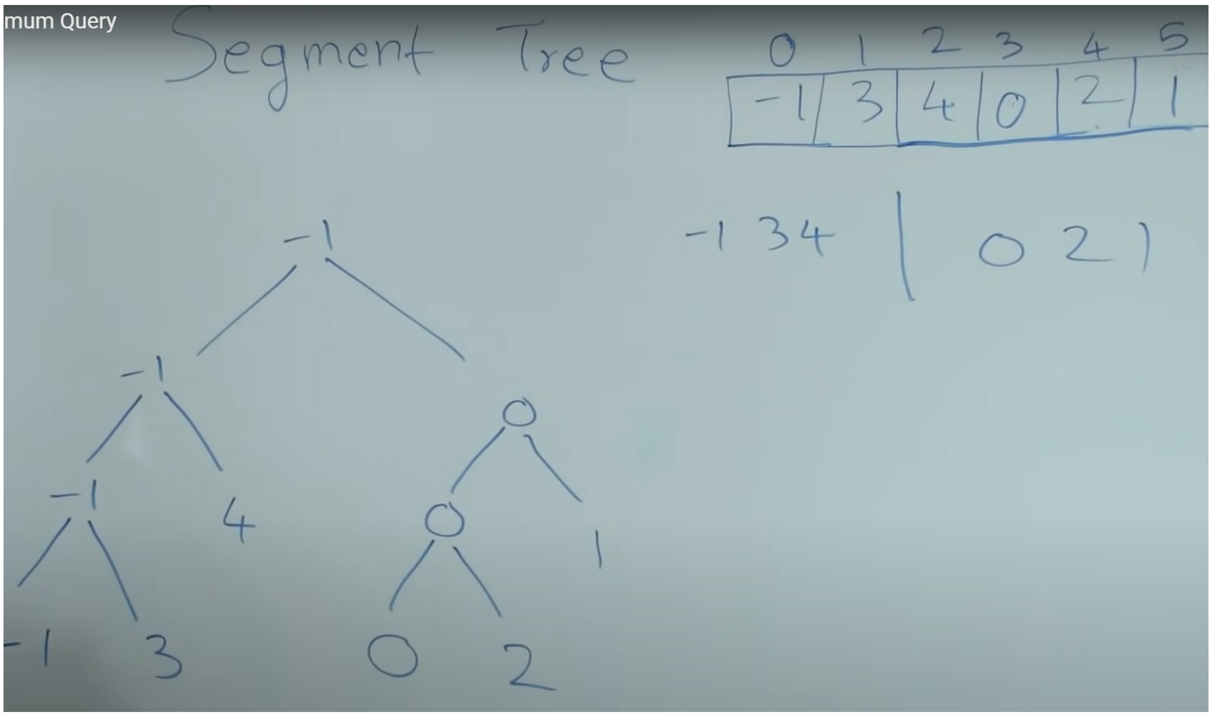
Another way could be using a multiset. The multiset implements a BST.

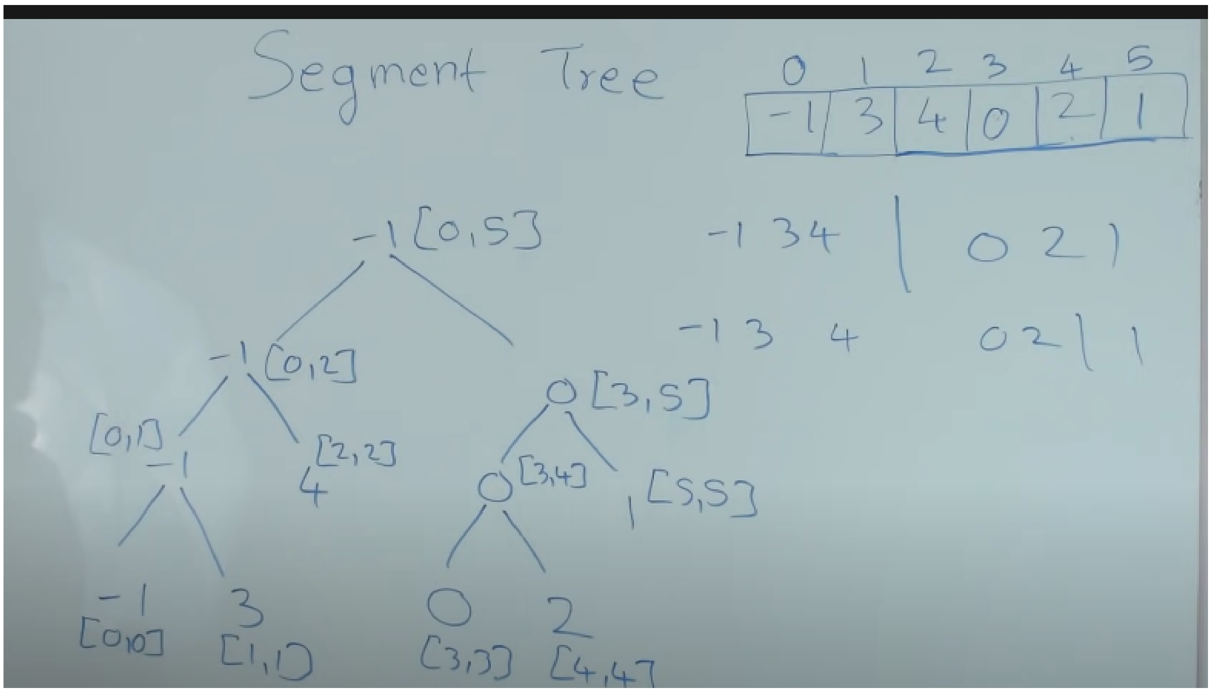
Segment Tree

It helps us answer questions of intervals.
Min number in an interval.
Max number in an interval.
Sum of an interval.

you could answer these questions using a matrix but it would take O(n^2) to build the matrix.
O(n) to build the segment tree.
Answer in O(Log n)

A segment tree is a binary tree.
The elements of the array will be the leafs of the binary tree.
To build the tree split the array in half.





Does my query range overlap the range of the current node?
Partial overlap: If the overlap is partial then we are going to keep searching for both sides of the subtree.
Total overlap: Stop. When the current node interval is contained on the query interval then this is one part of the answer. We return the value of this node.
No overlap: Stop and return a really big number. In this video he returned a max because it is the opposite thing of what we are asking.

The size of the vector<int> tree:

1. if(n == a power of two) tree.resize((2 * n) - 1);
2. else{//Encontramos la siguiente potencia de dos.
```
int next = pow(2, ceil(log(n) / log(2)));//Get next power of two.
        tree.resize((2 * next) - 1);
```
   }

Function to build the tree from an array.
build(0, n - 1, 0);
x = current node, lx = left of current interval, rx = right of current interval.
```
void build(int l, int r, int x){
    if(l == r){//Leaf node.
        tree[x] = a[l];
        return;
    }
    int m = (l + r) / 2;//Mid
    build(l, m, (x * 2) + 1);//Left subtree
    build(m + 1, r, (x * 2) + 2);//Right subtree
    tree[x] = tree[(x * 2) + 1] + tree[(x * 2) + 2];//Actual node is
sum of both children.
}
```
Function to calculate a query
sum(i, v - 1, 0, 0, n - 1);
l and r are the interval of the query.
```
int sum(int l, int r, int x, int lx, int rx){
    //if(l > r) return 0;
    if(lx >= l && rx <= r) return tree[x];
    //Check if you are including the r element of not.
    if(lx > r || l > rx) //lx > r || l > rx will include the r element
on the sum.
        return 0;
    int m = (lx + rx) / 2;
    int s1 = sum(l, r, (2 * x) + 1, lx, m);
    int s2 = sum(l, r, (2 * x) + 2, m + 1, rx);
    return s1 + s2;
}
```

Function to make an update
set(i, v, 0, 0, n - 1);
i = pos of the Tree array to update. v value
```
void set(int i, int v, int x, int lx, int rx){
    if(lx == rx) {
        tree[x] = v;
        return;
    }else{
        int m = (lx + rx) / 2;
        if(i <= m) set(i, v, (x * 2) + 1, lx, m);
        else set(i, v, (x * 2) + 2, m + 1, rx);
        tree[x] = tree[(x * 2) + 1] + tree[(x * 2) + 2];
    }
}
```

BST

A binary search tree (BST), a special form of a binary tree, satisfies the binary search property:

1. The value in each node must be greater than (or equal to) any values stored in its left subtree.
2. The value in each node must be less than (or equal to) any values stored in its right subtree.

Los valores del subarbol de la derecha deben ser mayores o igual que su padre y los de la izquierda menor o igual que su padre.

Valid BST

```
class Solution {
private:
   bool isBST(TreeNode* root, long long min, long long max){
      if(root == nullptr) return true;
      long long value = root->val;
      if(value < min || value > max) return false;

      bool ans_izq = isBST(root->left, min, value - 1);
      bool ans_der = isBST(root->right, value + 1, max);
      return ans_izq && ans_der;
   }

public:
   bool isValidBST(TreeNode* root) {
      long long min = numeric_limits<int>::min();
      long long max = numeric_limits<int>::max();
      return isBST(root, min, max);
   }
};
```

If we do inorder traversal, we can get the data from the tree in ascending order.

We are going to use a stack to save the data from least to greatest.

Operations

Search

```
TreeNode* searchBST(TreeNode* root, int val) {
      if(root == nullptr) return nullptr;
      if(root->val == val) return root;

      if(val < root->val) return searchBST(root->left, val);

      return searchBST(root->right, val);
   }
```

Si estamos en el nodo que buscamos, lo regresamos. Si el valor a buscar es menor que el valor de nuestro nodo entonces buscamos por la izquierda.
Buscamos por la derecha.

Insert

```
TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(root == nullptr) return new TreeNode(val);
        if(val < root->val) root->left = insertIntoBST(root->left, val);
        else root->right = insertIntoBST(root->right, val);
        return root;
    }
```

KTH largest

```
class KthLargest {
private:
    struct Node{
        int value;
        Node* left, right;
    }

    Node* delete_min(Node* root){
        Node* parent = nullptr;
        Node* node = root;
        while(node->left != nullptr){//Mientras no lleguemos al final
            parent = node;
            node = node->left;
        }
        Node* replacement = node->right;
        Node* new_root;
        if(parent == nullptr) new_root = replacement;//We are in the root.
        else {
            parent->left = replacement;
            new_root = root;
        }
        delete node;
        return new_root;
    }

    int get_min(Node* root){
        Node* node = root;
        while(node->left != nullptr)
            node = node->left;
        return node->value;
    }

    Node* add(Node* root, int value){
        if(root == nullptr) return new Node{value, nullptr, nullptr};
        if(value < root->value) root->left = add(root->left, value);
        else root->right = add(root->right, value);
        return root;
    }
    Node* root = nullptr;
    int size = 0, k;
public:
    KthLargest(int k, vector<int>& nums) {
        k = k;
        for(int i = 0; i < nums.size(); i++){
            root = add(k, nums[i]);
            size++;
            if(size > k){
                root = delete_min(root);
                size--;
            }
        }
    }

    int add(int val) {
        root = add(root, val), size++;
        if(size > k) root = delete_min(root), size--;
        int kth = get_min(root);
        return kth;
    }
};

 * Your KthLargest object will be instantiated and called as such:
 * KthLargest* obj = new KthLargest(k, nums);
 * int param_1 = obj->add(val);
```

Gráficas

Representación

Matriz de Adyacencia
Consulta O(1).
Espacio O(n^2).
Calcular el grado O(n).
Encontrar vecino O(n).

Lista de adyacencia
Espacio O(n + m).
Calcular el grado O(1).
Encontrar vecino O(vecinos).
Saber si dos vértices estan conectados O(vecinos).

BFS

O(V + E). En Adjacency List.
O(v * E). En Adjacency Matrix.
1.Agrega un vértice a la cola.
2. Mientras la cola no este vacía.
   a. Selecciona el siguiente vértice en la cola.
   b. Agrega todos sus vecinos a la cola (que no hayan sido agregados).
   c. Elimina de la cola.

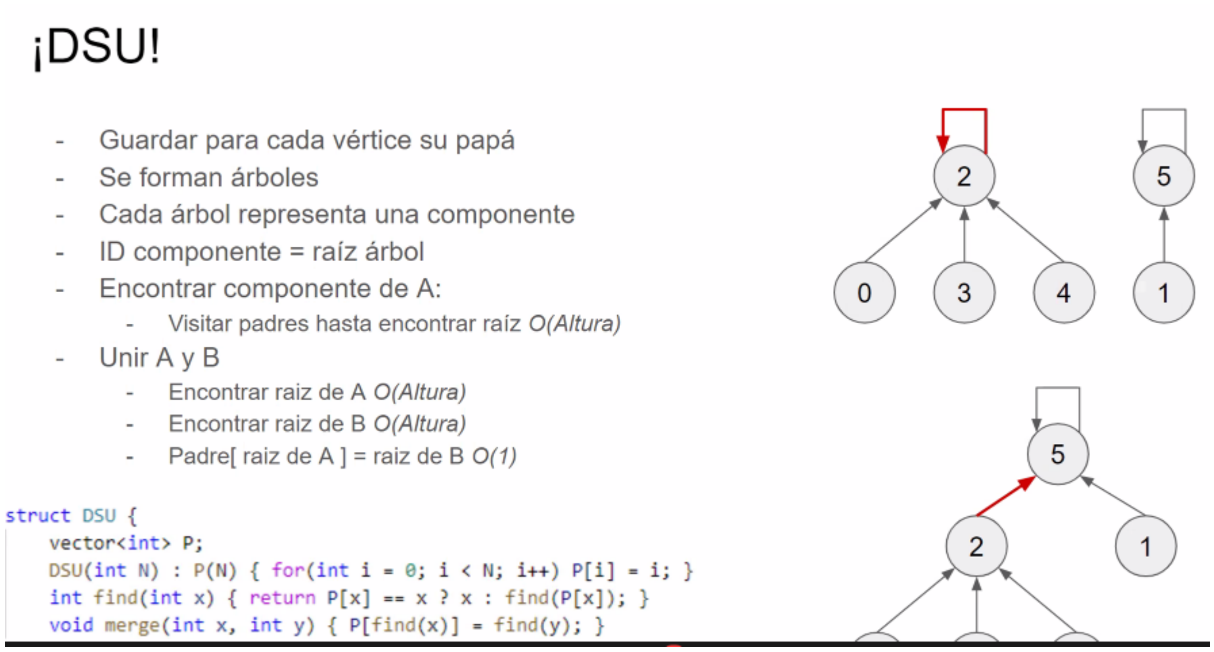
```
void BFS(vector<vector<int>>& Graph, int start, vector<bool> &visited) {
        queue<int> q;//Inicializamos nuestra cola.
        q.push(start), visited[start] = true;//Le agregamos el nodo start, le ponemos que lo
visitamos.
        while (!q.empty()) {//Mientras la cola no este vacia.
                int node = q.front();//Agarramos el valor del nodo.

                for (int neighboor : Graph[node]) {//Recorremos los vecinos del nodo actual.
                        if (!visited[neighboor])//Si no lo hemos visitado.
                                q.push(neighboor), visited[neighboor] = true;//Agregamos los
vecinos a la cola y ponemos que ya los visitamos.
                }
                q.pop();//Sacamos el nodo de la cola.
        }
}
```

DSU

Te permite agregar una arista entre dos vértices.
Determinar si dos vértices pertenecen a la misma componente.

Cada árbol es un componente.
Id componente es la raíz del árbol.
Guardar para cada vértice su papá
Encontrar componente de A:
    -Visitar padres hasta encontrar la raíz O(Altura).
Unir A y B.
- Encontrar raíz de A O(Altura).
- Encontrar la raíz de B O(Altura).
- Padre[raíz de A] = raíz de B O(1).

DSU



¡DSU!

- Guardar para cada vértice su papá
- Se forman árboles
- Cada árbol representa una componente
- ID componente = raíz árbol
- Encontrar componente de A:
    - Visitar padres hasta encontrar raíz *O(Altura)*
- Unir A y B
    - Encontrar raiz de A *O(Altura)*
    - Encontrar raiz de B *O(Altura)*
    - Padre[raiz de A] = raiz de B *O(1)*

```
struct DSU {
    vector<int> P;
    DSU(int N) : P(N) { for(int i = 0; i < N; i++) P[i] = i; }
    int find(int x) { return P[x] == x ? x : find(P[x]); }
    void merge(int x, int y) { P[find(x)] = find(y); }
```

Code:

```
struct DSU {
        vector<int> P;
        DSU(int N) : P(N) { for(int i = 0; i < N; i++) P[i] = i; }
        int find(int x) { return P[x] == x ? x : find(P[x]); }
        void merge(int x, int y) { P[find(x)] = find(y); }
};
```

Union by Rank

O (Log n)
Adjunta la componente más chica a la más grande. Cuando tienen la misma altura, conecta la componente de x a y.
Observación 1:
Si altura(A) > altura(B)
        Podemos conectar B hacia A y altura(A) no cambia.

Observación 2:
Si altura(A) == altura(B)
        altura(A) aumenta en 1.

Code:

```
struct DSU {
        vector<int> P, H;
        DSU(int N) : P(N), H(N, 0) { for(int i = 0; i < N; i++) P[i] = i; }
        int find( int x ) { return P[x] == x ? x : find(P[x]); }
```

```
	void merge(int x, int y) {
		int raiz_x = find(x);
		int raiz_y = find(y);
		if( H[raiz_x] > H[raiz_y] ) swap(raiz_x, raiz_y);
		P[raiz_x] = raiz_y;
		if( H[raiz_x] == H[raiz_y] ) H[raiz_y]++;
	}
};
```

Path Compression

O(log N) amortized.
No pasa nada si lo recorremos todo una vez, el problema es recorrerlo todo en cada consulta.
Todos los vértices que recorramos, redirigir a root.

Code:

```
struct DSU {
	vector<int> P;
	DSU( int N ) : P(N) { for(int i = 0; i < N; i++) P[i] = i; }
	int find( int x ) { return P[x] == x ? x : P[x] = find(P[x]); }
	void merge( int x, int y ) { P[find(x)] = find(y); }
};
```

Dijkstra

O(E log E)

Distancias positivas.

Usamos multiset para procesar las aristas con menor peso.

Considerar eventos “Llegué a un vértice”
	-Tiempo de llegada
	-Vértice de llegada
Procesar eventos en orden de tiempo
	-Revisar que sea un vértice inexplorado
		-Usar aristas para agregar nuevos eventos

Code:

```
#include <limits>
#define INF LLONG_MAX

struct Edge { long long to, w; };
bool operator <(const Edge& a, const Edge& b) { return a.w < b.w; }
int main() {
	std::ios_base::sync_with_stdio(0);
	std::cin.tie(0);
	long long n = 0, m = 0;
	std::cin >> n >> m;
	std::vector<std::vector<Edge>> g(n + 1);

	long long a = 0, b = 0, c = 0;
	Edge aux;

	for (int j = 0; j < m; j++){ //Read the graph.
		std::cin >> a >> b >> c; // a is from.
		aux.to = b;
		aux.w = c;
		g[a].push_back(aux); // One way graph.
	}

	std::multiset<Edge> edges;
	std::vector<long long> Dist(n + 1, INF); //Initalize distances in INF.
	edges.insert({ 1, 0 });
	while (!edges.empty()) { //While there are edges.
		auto [node, dist] = *edges.begin();
		edges.erase(edges.begin());
		if (Dist[node] != INF) continue;
		Dist[node] = dist;
		for (auto edge : g[node])  //We traverse the neighbors of the current node
			edges.insert({ edge.to, edge.w + dist });
	}

	for (unsigned int i = 1; i < n + 1; i++)
		std::cout << Dist[i] << " ";
}
```

MST

Prim

O(N Log N).
Empezamos a recorrer el árbol desde el primer vértice y luego, agarramos la menor que sale de todo el arbol.

Code:

```
struct Edge { int to = 0, weight = 0; };

//Sobrecarga de operadores.
```

```
bool operator <(const Edge& a, const Edge& b) { return a.weight < b.weight; }

int main() {
    int n = 0, m = 0, a = 0, b = 0, c = 0;
    cin >> n >> m;

    vector<vector<Edge>> Aristas(n + 1, vector<Edge>(n + 1));

    for (int i = 0; i < n; i++) { //Reading from, to, weight.
        cin >> a >> b >> c;
        Edge edge; // Instantiate Edge object..
        edge.to = b; edge.weight = c;
        Aristas[a][b] = edge;
        edge.to = a;
        Aristas[b][a] = edge;//Two ways roads.
    }
// Multiset to have always the next smallest edge.
    multiset<Edge> edges; vector<bool> vis(n + 1, false);
    long long ans = 0;
    edges.insert({ 1, 0 }); // Initialize edges.
    while (!edges.empty()) { // Mientras haya vértices.
        auto [node, w] = *edges.begin(); //Access the cheaper road.
        edges.erase(edges.begin()); //Eliminate the cheapest because we already processed it.
        if (vis[node]) continue; //Si ya lo visitamos, continue.
        ans += w; //Actualize the minimum weight of the graph.
        vis[node] = true; //Visited
        for (Edge e : Aristas[node]) edges.insert(e); //Add the edges connected to the actual
node.
    }

    cout << ans;
    return 0;
}
```

Kruskal

Vamos agarrando las aristas más pequeñas, si no forman ciclos.
Para ver si forman un ciclo, usamos DSU.

O(V + ElogV).
E log v para ordenar aristas. V del dsu.

Code:

```
struct DSU{
        std::vector<int> P;
```

```
	DSU(int N) : P(N) {
		for (int i = 0; i < N; i++)
			P[i] = i;
	}
	int find(int x) {//Te regresa tu componente
		return P[x] == x ? x : find(P[x]);
	}
	void merge(int x, int y) {
		P[find(x)] = find(y);
	}
};

struct Edge{
	int w, from, to;
};

bool operator <(const Edge& a, const Edge& b) { return a.w < b.w; }

int main() {//n vertex and m edges.
	std::ios_base::sync_with_stdio(0);
	std::cin.tie(0);
	int n, m, a, b, c, k = 0;

	std::cin >> n >> m;
	std::vector<Edge> Edges;

	Edge edge;

	for (int i = 0; i < m; i++){
		std::cin >> a >> b >> c;
		edge.from = a;
		edge.to = b;
		edge.w = c;
		Edges.push_back(edge);
	}

	sort(Edges.begin(), Edges.end());//Ordenamos las aristas de menor a mayor.
	DSU dsu(n + 1);
	long long ans = 0;
	for(Edge e: Edges)//Recorremos todos las aristas.
		if (dsu.find(e.from) != dsu.find(e.to)) { //Si la componente de from no es igual a
to.
			dsu.merge(e.from, e.to);
			k++;
			ans += e.w;
		}

	if(k == n - 1) std::cout << ans;
```

```
        else std::cout << "IMPOSSIBLE";

        return 0;
}*/
```

Bellman Ford

O(V * E)
Acepta números negativos.

//for de 0 a v - 1 para recorres los vertices
//for anidado de 0 que va a pasar por toda tu lista de adyacencia.
//Mínimo entre tu distancia actual o la distancia en a + el peso.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#define INF 100000000000LL

using namespace std;

struct Edge{
   int from, to, weight;
};

int main(){
   std::ios_base::sync_with_stdio(0);
        std::cin.tie(0);
   short n = 0, m = 0;
   int a = 0, b = 0, c = 0;
   cin >> n >> m;

   vector<Edge> edges(m);

   for(int i = 0; i < m; i++){
      cin >> a >> b >> c;
      edges[i] = { a, b, c};
   }

   vector<int> Dist(n + 1, INF);
   Dist[0] = 0;//El primer vertice nos cuesta 0.
   for( int i = 0; i < n - 1; i++){//Por cada vertice.
      for(int j = 0; j < m; j++ ){//Recorres sus aristas.
         int from = edges[j].from; //From
         int to = edges[j].to; //To
```

```
            int weight = edges[j].weight;
            if(Dist[from] != INF){//Si de donde salimos no es INF.
                //Actualizamos Dist[to]
                Dist[to] = min(Dist[to], Dist[from] + weight);//min(a donde vamos y de donde
salimos + el peso)
            }
        }
    }
    //Check for cycles
    /*for(int i = 0; i < edges.size(); i++){
        int u = edges[i].from;
        int v = edges[i].to;
        if(Dist[u] != INF && Dist[v] > Dist[u] + edges[i].weight){//Si encontramos otra distancia
menor es porque hay un ciclo
            cout << "There is a cycle\n";
            break;
        }
    }*/

    for(int i = 1; i <= n; i++)
        cout << Dist[i] << " ";
    return 0;
}
```

Floyd Warshall

O(V^3)
Nos da todas las distancias a todos los demás vértices.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

#define INF 10000000000000LL

int main() {
        int q = 0, n = 0, m = 0, a = 0, b = 0, c = 0;
        cin >> n >> m >> q;

        vector<vector<long long>> Matrix(n + 1, vector<long long> (n + 1, INF));

        for (int i = 1; i <= n; i++)
                Matrix[i][i] = 0;
```

```
	for (int i = 0; i < m; i++) {
		cin >> a >> b >> c;
		if (Matrix[a][b] > c) {
			Matrix[a][b] = c;
			Matrix[b][a] = c;
		}
	}

	for (int i = 1; i <= n; i++){//El origen
		for (int j = 1; j <= n; j++){//De donde voy
			for (int k = 1; k <= n; k++) {//A donde
				if(Matrix[j][k] > Matrix[j][i] + Matrix[i][k])
					Matrix[j][k] = Matrix[j][i] + Matrix[i][k];
			}
		}
	}

	for (int i = 0; i < q; i++){
		cin >> a >> b;
		Matrix[a][b] != INF ? cout << Matrix[a][b] << "\n" : cout << "-1\n";
	}

	return 0;
}
```

DFS

```
void dfs(int x, vector<vector<int>> &g, vector<bool> &vis){
    vis[x] = true;
    for(int y : g[x]){
        if(y == x) continue;
        if(!vis[y]) dfs(y, g, res);
    }
}
```

Topo Sort

O(V + E)
Node A goes before node B. DAG.
Code:

```
vector<bool> vis, rStack;
vector<vector<int>> g;
vector<int> ans;
bool cycle = false;
bool dfs(int);
```

```
for(int i = 1; i <= n; i++){
        if(!vis[i]) if(dfs(i)) break;
    }

if(cycle){
        cout << "IMPOSSIBLE\n";
        return 0;
    }

    reverse(ans.begin(), ans.end());
    for(int i = 0; i < ans.size(); i++) cout << ans[i] << " ";

    return 0;


bool dfs(int x){
    vis[x] = true;
    rStack[x] = true;
    for(int y : g[x]){
        if(!vis[y])
            if(dfs(y)) return true;
        if(rStack[y]){
            cycle = true;
            return true;
        }
    }
    rStack[x] = false;
    ans.push_back(x);
    return false;
}
```

Strongly Connected Components

Tarjan

Kosaraju

1. Perform DFS traversal of graph. Push node to Stack before returning.
2. Find the transpose graph by reversing the edges.
3. Pop nodes one by one from stack and again do DFS on modified graph.(Keep popping nodes). Each successful DFS gives 1- SCC.
4.

Code:

```
#include <bits/stdc++.h>
```

```
using namespace std;
const int maxN = 1e5+1;

bool vis[maxN];
int N, M, rt[maxN];
vector<int> ord, comp, G[maxN], GR[maxN];

void dfs1(int u){
    vis[u] = true;
    for(int v : G[u])
        if(!vis[v])
            dfs1(v);
    ord.push_back(u);
}

void dfs2(int u){
    vis[u] = true;
    comp.push_back(u);
    for(int v : GR[u])
        if(!vis[v])
            dfs2(v);
}

int main(){
    scanf("%d %d", &N, &M);
    for(int i = 0, a, b; i < M; i++){
        scanf("%d %d", &a, &b);
        G[a].push_back(b);//Normal Graph
        GR[b].push_back(a);//Transpose Graph
    }

    for(int i = 1; i <= N; i++)//1st step
        if(!vis[i])
            dfs1(i);

    int K = 0;
    fill(vis+1, vis+N+1, false);//Vis equals to false.
    reverse(ord.begin(), ord.end());//Topo sort or Stack
    for(int u : ord){
        if(!vis[u]){
            dfs2(u);//3rd Step DFS on modified graph.
            K++;
            for(int v : comp)//Print the numbers of nodes in a component.
                rt[v] = K;
            comp.clear();
        }
    }
    printf("%d\n", K);
    for(int i = 1; i <= N; i++)
        printf("%d%c", rt[i], (" \n")[i==N]);
}
```

Sorting

Quick Sort

O(n log n)

```
int partition(vector<int>& v, int low, int high){
    int i = low;
    int j = high;
    int pivot = v[low];
    while (i < j){
        while (pivot >= v[i])//ItemFromLeft bigger than pivot
            i++;
        while (pivot < v[j])//ItemFromRight smaller than pivot
            j--;
        if (i < j)
            swap(v[i], v[j]);
    }
    swap(v[low], v[j]);
    return j;
}

void quicksort(vector<int> & v, int low, int high){
    if(low < high){
        int pivot = partition(v, low, high);
        quicksort(v, low, pivot - 1);
        quicksort(v, pivot + 1, high);
    }
}
```

Merge Sort

O (n log n)

```
void merge(vector<int>& v,int inicio, int mitad, int final){
    int elementosIzq = mitad - inicio + 1;//Tamaño de nuestro vector 
izquierda.
    int elementosDer = final - mitad;//Tamaño de nuestro vectro derecha

    vector<int>izq(elementosIzq);
    vector<int>der(elementosDer);
    for(int i = 0; i < elementosIzq; i++){//Agregar los elementos de la
izquierda.
        izq[i] = v[inicio + i];
    }
    for(int j = 0; j < elementosDer; j++){//Agregar los elementos de la
derecha.
        der[j] = v[mitad + 1 + j];
    }

    int i = 0, j = 0, k = inicio;

    while(i < elementosIzq && j < elementosDer){//Comparación, para ir
ordenando el vector.
        if(izq[i] <= der[j]){//Si el de la izquierda es menor, lo
agregamos, i++.
            v[k] = izq[i];
            i++;
        }else{
            v[k] = der[j];
            j++;
        }
        k++;
    }

    //Añadir los elemento faltantes.
    while(j < elementosDer){
        v[k] = der[j];
        j++;
        k++;
    }

    while(i < elementosIzq){
        v[k] = izq[i];
        i++;
        k++;
    }
}

void mergeSort(vector<int>& v,int inicio, int final){
    if(inicio < final){
        int mitad = inicio + (final - inicio)/2;
        mergeSort(v, inicio, mitad);
        mergeSort(v, mitad + 1, final);
        merge(v, inicio, mitad, final);
    }
}
```

Dynamic Programming

Greedy algorithms

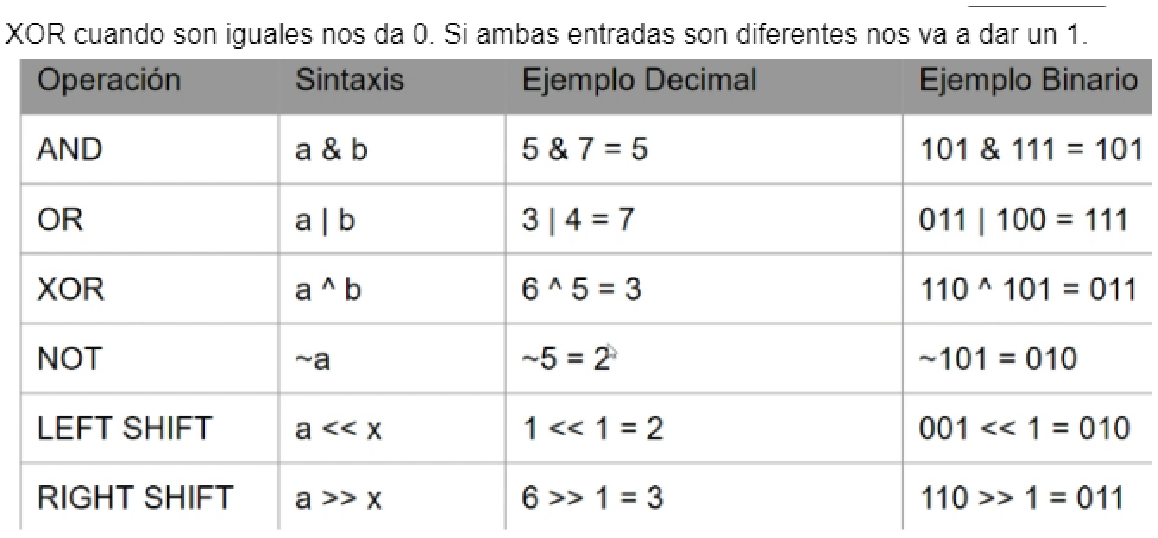
Fractional Knapsack problem.

Bitwise operators

Operators that work with binary.

&, |, <<, >>, ~, ^ Bitwise operator

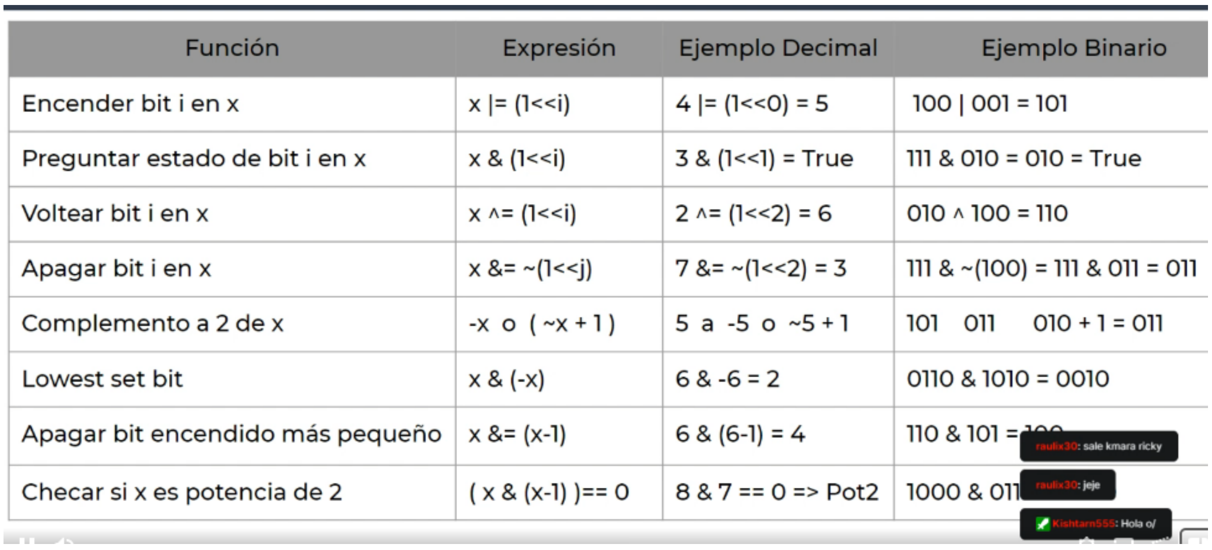
A ^ B es lo mismo que escribir A != B



XOR cuando son iguales nos da 0. Si ambas entradas son diferentes nos va a dar un 1.

Operación	Sintaxis	Ejemplo Decimal	Ejemplo Binario
AND	a & b	5 & 7 = 5	101 & 111 = 101
OR	a | b	3 | 4 = 7	011 | 100 = 111
XOR	a ^ b	6 ^ 5 = 3	110 ^ 101 = 011
NOT	~a	~5 = 2	~101 = 010
LEFT SHIFT	a << x	1 << 1 = 2	001 << 1 = 010
RIGHT SHIFT	a >> x	6 >> 1 = 3	110 >> 1 = 011

El shift recorre todos sus bits a la izquierda
Inserta 0 a la derecha y lo que este hasta la izquierda se va a perder.
shift right
Recorre sus bits a la derecha y hasta el final inserta el 0.
inserta 0 a la izquierda y lo que este hasta la derecha se va a perder.



Función	Expresión	Ejemplo Decimal	Ejemplo Binario
Encender bit i en x	x |= (1<<i)	4 |= (1<<0) = 5	100 | 001 = 101
Preguntar estado de bit i en x	x & (1<<i)	3 & (1<<1) = True	111 & 010 = 010 = True
Voltear bit i en x	x ^= (1<<i)	2 ^= (1<<2) = 6	010 ^ 100 = 110
Apagar bit i en x	x &= ~(1<<j)	7 &= ~(1<<2) = 3	111 & ~(100) = 111 & 011 = 011
Complemento a 2 de x	-x o (~x + 1)	5 a -5 o ~5 + 1	101 011 010 + 1 = 011
Lowest set bit	x & (-x)	6 & -6 = 2	0110 & 1010 = 0010
Apagar bit encendido más pequeño	x &= (x-1)	6 & (6-1) = 4	110 & 101 =
Checar si x es potencia de 2	(x & (x-1))== 0	8 & 7 == 0 => Pot2	1000 & 011

raulix30: sale kmara ricky

raulix30: jeje

Kishtarn555: Hola o/

Math

Binary Exponentiation

O(log n)
Without modulo.

Code:

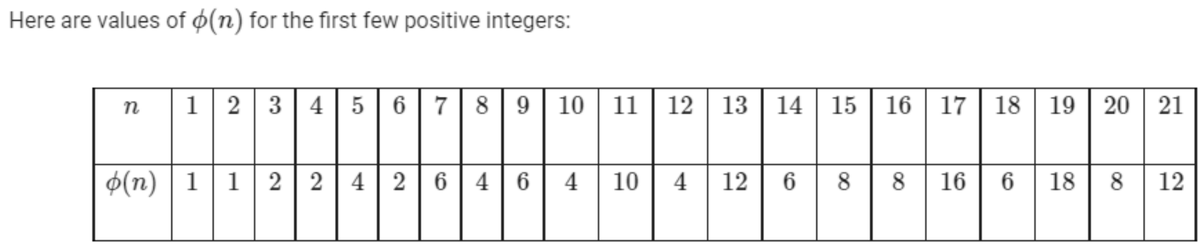
```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

Problem: Compute x^n mod m.

Code:

```
long long binpow(ll a, ll b, ll mod) {
    a %= mod;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return res;
}
```

Euler Totient Function



Here are values of $\phi(n)$ for the first few positive integers:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\phi(n)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8	12

O(sqrt(n))

Code:

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}
```

O(n log log n)

Code:

```
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

Counting Divisors

Code:

```
//Resolvemos para todos los X, y ya después vemos las consultas
//Serie armonica, crece como lógaritmo de n
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main(){
    long long n = 0, aux = 0;
    cin >> n;

    long long max = 1000000 +1;
    vector<int> nums_div(max);

    for(int i = 1; i < max; i++){//Recorremos hasta el máx
        for(int j = i; j < max; j += i){//Recorremos hasta el max, a partir de nuestro divisor actual,
aumentamos al sig divisor.
            nums_div[j]++;
        }
    }

    for(int i = 0; i < n; i++){
        cin >> aux;
        cout << nums_div[aux] << " ";
    }

    return 0;
}
```