Codigo tem que ser por metro - ICPC Library

Contents

1 Data Structures

1.1 DSU (Disjoint Set Union)

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const int N = 10; // Number of vertex

// Disjoint Set Union implementation using array (0-Based)
int parent[N];

// Array to Union by Size (0-Based)
int size[N];

// Function used to initialize Disjoint Set
void Build(){
    for(int i = 0; i < N; i++){
        parent[i] = i;
        size[i] = 1;
    }
}

// Returns the representative of the set that contains the element "v" (
    Path Compression Optimization)
int Find(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = Find(parent[v]);
}

// Joins two different sets (Union by Size Optimization)
void Union(int a, int b) {
    a = Find(a);
    b = Find(b);
    if (a != b) {
        // We put the smallest set in the largest
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}

/*

Time Complexity

Build   -> O(N)
Find    -> O(logN) ( In the worst case, the average case is O(1) )
Union   -> O(logN) ( In the worst case, the average case is O(1) )

Links:

https://cp-algorithms.com/data_structures/disjoint_set_union.html
https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/

*/
```

1.2 BIT (Fennwick Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ll = long long;
```

```cpp
// Implementation of a Binary Indexed Tree (Fennwick Tree) (1-Based) (Sum
    Operation)
vector<int> bit;

// Array used to construct BIT (0-Based)
vector<int> v;

// Initialize Fennwick Tree
void build(){
    // Initialize all values with 0 (1-based)
    bit = vector<int>(v.size()+1, 0);

    // Putting the values of "v" in bit
    for(int i = 0; i < v.size(); i++) bit[i+1] = v[i];

    // Updating the values
    for(int i = 1; i < bit.size(); i++){
        int idx = i + (i & (-i));
        if(idx < bit.size()) bit[idx] += bit[i];
    }
}

// Return the sum of [0,idx] in "v"
int prefix_query(int idx){
    int result = 0;
    for(++idx; idx > 0; idx -= idx & -idx) result += bit[idx];
    return result;
}

// Computes the range sum between two indices (both inclusive) [l,r] in "v"
int range_query(int l, int r){
    if (l == 0) return prefix_query(r);
    else return prefix_query(r) - prefix_query(l - 1);
}

// Update bit adding "add" (idx represent the position in "v")
void update(int idx, int add) {
    for (++idx; idx < bit.size(); idx += idx & -idx) bit[idx] += add;
}

/*

Time Complexity

build           -> O(n)
prefix_query    -> O(logn)
range_query     -> O(logn)
update          -> O(logn)

Links:

https://www.youtube.com/watch?v=uSFzHCZ4E-8
https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/
https://cp-algorithms.com/data_structures/fenwick.html
https://www.youtube.com/watch?v=v_wj_mOAlig
https://www.youtube.com/watch?v=CWDQJGaN1gY

*/
```

1.3 BIT 2D (Fennwick Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ll = long long;

// Implementation of a Fennwick Tree 2D (1-Based) (Sum Operation)
vector<vector<int>> bit;
// Matrix used to construct BIT (0-Based)
vector<vector<int>> v;

void build(){ // Initialize Fennwick Tree 2D
    bit.assign(v.size()+1, vector<int>(v.size()+1, 0)); // Bit[N+1][N+1]

    for(int i = 0; i < v.size(); i++)
        for(int j = 0; j < v[i].size(); j++)
            bit[i+1][j+1] = v[i][j];

    for(int i = 1; i < bit.size(); i++){
        for(int j = 1; j < bit[i].size(); j++){
            int idx_i = i + (i & (-i));
            int idx_j = j + (j & (-j));
            if(idx_i < bit.size() && idx_j < bit[i].size())
                bit[idx_i][idx_j] += bit[i][j];
        }
    }
}

// Returns the sum of [0,0] to [i,j]
int query(int i, int x){
    int result = 0;
    for(++i; i > 0; i -= i & -i)
        for(int j = x+1; j > 0; j -= j & -j)
            result += bit[i][j];
    return result;
}

// Returns the sum of (i_1,j_1) to (i_2,j_2) (both inclusive)
int range_query(int i_1, int j_1, int i_2, int j_2){
    return query(i_2,j_2) - query(i_1-1,j_2) - query(i_2,j_1-1) + query(i_1
        -1,j_1-1);
}

// Update bit adding "add" to position v[i][x]
void update(int i, int x, int add){
    for(++i; i < bit.size(); i += i & -i)
        for(int j = x+1; j < bit[i].size(); j += j & -j)
            bit[i][j] += add;
}

/*

Time Complexity

build       -> O(n^2)
query       -> O((logn)^2)
range_query -> O((logn)^2)
update      -> O((logn)^2)

Links:

https://cp-algorithms.com/data_structures/fenwick.html#finding-minimum-of
    -0-r-in-one-dimensional-array

*/
```

1.4 Segment Tree

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const int MAX = 20000; // Size of v (Array used to construct)

// Implementation of a Recursive Segment Tree (1-Based) (Sum Operation)
ll segTree[4*MAX];

// Array used to construct Segtree (Need to be 1-Based)
vector<ll> v;
```

```
// SINGLE-ELEMENT MODIFY, RANGE QUERY [l,r]

// Call using build(1,1,N) (N = v.size() - 1)
void build(int p, int l, int r){

    // Building Leaf
    if(l == r){
        segTree[p] = v[r];
    }
    else{

        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        // Building Left Children
        build(lc,l,m);

        // Building Rigth Children
        build(rc,m+1,r);

        // Building Node (Sum Operation)
        segTree[p] = segTree[lc] + segTree[rc];

    }
}

// Call using update(1,1,N,idx,value) (idx need to be 1-Based)
void update(int p,int l, int r, int idx, ll value){

    // Updating Leaf
    if(l == r){
        segTree[p] = value;
    }
    else{

        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        // Modified Leaf is on the left.
        if(idx <= m){

            update(lc,l,m,idx,value);
        }
        // Modified Leaf is on the rigth.
        else{

            update(rc,m+1,r,idx,value);
        }

        // Update Node
        segTree[p] = segTree[lc] + segTree[rc];

    }
}

// Call using query(1,1,N,ql,qr) (ql and qr need to be 1-Based)
ll query(int p, int l, int r, int ql, int qr){

    // This node is inside the range answer
    if(ql <= l && r <= qr){
        return segTree[p];
    }
    else{

        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        // Our answer is just in the Left Children
        if(qr <= m){

            return query(lc,l,m,ql,qr);
        }
        // Our answer is just in the Rigth Children
        else if(ql > m){

            return query(rc,m+1,r,ql,qr);
        }
        else{

            // Our answer is an intersection of the 2 sides
            return query(lc,l,m,ql,qr) + query(rc,m+1,r,ql,qr);
        }
    }
}

// Range update [L,R]. It only works if the result converges. If there is a
      common value for the whole range [L,R] use Lazy Propagation.
void updateRange(int p, int l, int r, int ql, int qr){

    if(l == r){
        segTree[p] = value; // This value will vary according to each
             position of the "v" array
    }
    else{
        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        if(qr <= m){
            updateRange(lc, l, m, ql, qr);
        }
        else if(ql > m){
            updateRange(rc, m+1, r, ql, qr);
        }
        else{
            updateRange(lc,l,m,ql,qr);
            updateRange(rc,m+1,r,ql,qr);
        }
        segTree[p] = segTree[lc] + segTree[rc];
    }
}

/*

Time Complexity

build       -> O(n)
update      -> O(logn)
query       -> O(logn)
updateRange -> O(n)

Links:

https://cp-algorithms.com/data_structures/segment_tree.html

*/
```

1.5 SegTree with Lazy Propagation

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

#define int long long
using ll = long long;

// Array used to construct Segtree (Need to be 1-Based)
vector<int> v;

// Implementation of a Recursive Segment Tree with Lazy Propagation (1-
     Based) (Sum Operation)
// RANGE UPDATE [l,r], RANGE QUERY [l,r]
class segTree{
```

```
    vector<int> seg;
    vector<int> lazy;

public:
    // n = number of elements or v.size() - 1
    segTree(int n){
        seg.assign(4*n, 0);
        lazy.assign(4*n, 0);
        build(1,1,n);
    }
    // Call using update(1,1,n,ql,qr,value) (ql and qr need to be 1-Based)
    void update(int p, int l, int r, int ql, int qr, int value){
        propagate(p,l,r);
        if(r < ql || l > qr) return;
        if(ql <= l && r <= qr){
            lazy[p] = value;
            propagate(p,l,r);
        }
        else{
            int m = (l+r)/2;
            int lc = 2*p;
            int rc = lc + 1;

            update(lc,l,m,ql,qr,value);
            update(rc,m+1,r,ql,qr,value);

            seg[p] = seg[lc] + seg[rc];
        }
    }
    // Call using query(1,1,n,ql,qr) (ql and qr need to be 1-Based)
    int query(int p, int l, int r, int ql, int qr){
        propagate(p,l,r);
        if(r < ql || l > qr) return 0;
        if(ql <= l && r <= qr) return seg[p];

        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        return query(lc,l,m,ql,qr) + query(rc,m+1,r,ql,qr);
    }

private:

    void build(int p, int l, int r){
        if(l == r){
            seg[p] = v[l];
            lazy[p] = 0;
        }
        else{
            int m = (l+r)/2;
            int lc = 2*p;
            int rc = lc + 1;

            build(lc,l,m);
            build(rc,m+1,r);

            seg[p] = seg[lc] + seg[rc];
        }
    }

    void propagate(int p, int l, int r){
        if(lazy[p] == 0) return;

        seg[p] += (r-l+1)*lazy[p];
        if(l != r){
            lazy[2*p] += lazy[p];
            lazy[2*p + 1] += lazy[p];
        }
        lazy[p] = 0;
    }
};

/*

Time Complexity

build       -> O(n)
update      -> O(logn)
query       -> O(logn)

Links:

https://cp-algorithms.com/data_structures/segment_tree.html
https://www.youtube.com/watch?v=xuoQdt5pHj0
https://www.youtube.com/watch?v=UKH4Zgfa4kI
https://www.youtube.com/watch?v=3gPcs6PZPdk

*/
```

1.6 Policy Based Struct

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
        tree_order_statistics_node_update> indexed_set;

indexed_set s;

// Fun  o para inserir o elemento "X"
s.insert(X);

// Fun  o para remo  o do elemento "X"
s.erase(X);

// Retorna um iterator para o elemento na posi  o "X" (0-based indexing)
s.find_by_order(X);

// Retorna a posi  o do elemento "X", uma outra fun  o    contar a
    quantidade de elementos estritamente menores que "X"
s.order_of_key(X);


/*

Time Complexity

s.insert(X)          -> O(logn)
s.erase(X)           -> O(logn)
s.find_by_order(X)   -> O(logn)
s.order_of_key(X);   -> O(logn)

Observa  o: Quando o elemento n o existe no indexed_set a fun  o "
    order_of_key()" retorna a posi  o que ele DEVERIA ESTAR,
caso existisse, por isso     til para calcular a quantidade de elementos
    estritamente menores que "X".

Links:

https://codeforces.com/blog/entry/11080
https://www.geeksforgeeks.org/policy-based-data-structures-g/

*/
```

1.7 Multi Indexed Set

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

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

template<typename T>
class multi_indexed_set{

    tree<pair<T, int>, null_type, less<pair<T, int>>, rb_tree_tag,
            tree_order_statistics_node_update> miset;
    unordered_map<T, int> freq;

    public:
        void insert(T x){
            freq[x]++;
            miset.insert({x, freq[x]});
        }
        void erase(T x){
            if(!freq[x]) return;
            miset.erase({x, freq[x]});
            freq[x]--;
        }
        int order_of_key(T x){ return miset.order_of_key({x, 0}); }
        int count(T x) { return freq[x]; }
        int size(){ return miset.size(); }

};

multi_indexed_set<int> ms;

// F u n   o para inserir o elemento "X"
ms.insert(X);

// F u n   o para r e m o   o do elemento "X"
ms.erase(X);

// Retorna a p o s i   o do elemento "X", uma outra f u n   o     contar a
    quantidade de elementos estritamente menores que "X"
ms.order_of_key(X);

// Retorna a quantidade de elementos iguais a "X"
ms.count(X);

// Retorna a quantidade de elementos no multiset
ms.size();

/*

Time Complexity

ms.insert(X)        -> O(logn)
ms.erase(X)         -> O(logn)
ms.order_of_key(X); -> O(logn)
ms.count(X)         -> O(1) (Average)
ms.size()           -> ??

 O b s e r v a   o: Ainda    possivel a i m p l e m e n t a  o da f u n   o "
    find_by_order".

*/
```

2 Graph Algorithms

2.1 DFS

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

typedef long long ll;
const int N = 2000; // Number of vertices

// Graph inplementation using Adjacency List (0-Based)
vector<int> adj[N];

// Array to set visited Nodes
bool visited[N];

// Undirected Graph
void addEdge(int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// DFS
void dfsUtil(int v){

    // Mark current vertex as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    for(auto u: adj[v]){
        if(visited[u] == false){
            dfsUtil(u);
        }
    }

}

// Traverse all the Graph (Disconnected Graph) (If you know that is a
    connected Graph, just use "dfsUtil(root)")
void dfs(){

    // Set all unvisited
    for(int i = 0; i < N; i++){
        visited[i] = false;
    }

    // Visit all unvisited vertices
    for(int i = 0; i < N; i++){
        if(visited[i] == false){
            dfsUtil(i);
        }
    }

}

/*

Time Complexity

addEdge     -> O(1)
dfs         -> O(V+E)

Links:

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
https://www.geeksforgeeks.org/graph-implementation-using-stl-for-
    competitive-programming-set-1-dfs-of-unweighted-and-undirected/


*/
```

2.2 BFS

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

typedef long long ll;
const int N = 11; // Number of vertices
```

```cpp
// Graph inplementation using Adjacency List (0-Based)
vector<int> adj[N];

// Array to set visited Nodes
bool visited[N];

// Undirected Graph
void addEdge(int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// BFS
void bfsUtil(int v){

    queue<int> q;

    q.push(v);
    visited[v] = true;

    while(!q.empty()){

        int u = q.front();
        q.pop();

        for(auto i: adj[u]){

            if(visited[i] == false){
                q.push(i);
                visited[i] = true;
            }

        }
    }
}

// Traverse all the Graph (Disconnected Graph) (If you know that is a
     connected Graph, just use "bfsUtil(root)")
void bfs(){

    // Set all unvisited
    for(int i = 0; i < N; i++){
        visited[i] = false;
    }

    // Visit all unvisited vertices
    for(int i = 0; i < N; i++){
        if(visited[i] == false){
            bfsUtil(i);
        }
    }

}

// This function return true with a graph is bipartite
bool isBipartite(){

    // Store color of the vertex (-1 = unvisited, 0 = black, 1 = whites)
    vector<int> color(N, -1);

    // queue for BFS storing {vertex , colour}
    queue<pair<int, int> > q;

    //loop incase graph is not connected
    for (int i = 0; i < N; i++) {

        //if not coloured (not visited)
        if (color[i] == -1) {

            // Assign any color
            q.push({ i, 0 });
            color[i] = 0;

            //BFS
            while (!q.empty()) {

                pair<int, int> p = q.front();
                q.pop();

                //current vertex
                int v = p.first;
                //colour of current vertex
                int c = p.second;

                //traversing vertexes connected to current vertex
                for (int j : adj[v]) {

                    // Can't be bipartite
                    if (color[j] == c)
                        return false;

                    //if uncooloured (unvisited)
                    if (color[j] == -1) {
                        //colouring with opposite color to that of parent
                        color[j] = 1-c;
                        q.push({ j, color[j] });
                    }
                }
            }
        }
    }

    // Graph is Bipartite
    return true;
}

/*

Time Complexity

addEdge     -> O(1)
bfs         -> O(V+E)
isBipartite -> O(V+E)

Links:

https://www.geeksforgeeks.org/bfs-using-stl-competitive-coding/
https://www.geeksforgeeks.org/bipartite-graph/

*/
```

2.3 Topologial Sort

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const int N = 1e5+10; // Number of vertex

// Graph inplementation using Adjacency List (0-Based)
vector<int> adj[N];

// Directed Graph
void addEdge(int u, int v){
    adj[u].push_back(v);
}

// K a h n s algorithm for Topological Sorting (Directed Acyclic Graphs)
void topologicalSort(){

    // Stores the amount of edges reaching the vertex
    vector<int> inDegree(N, 0);
```

```
    // Calculating in degree O(V+E)
    for(int u = 0; u < N; u++) {
        for(auto v : adj[u]){
            inDegree[v]++;
        }
    }

    queue<int> q;

    // We need to start at a vertex with in degree = 0
    for(int i = 0; i < N; i++){
        if(inDegree[i] == 0) {
            q.push(i);
        }
    }

    // Number of vertices visited
    int cnt = 0;

    vector<int> answer;

    // BFS Traversal
    while (!q.empty()) {

        int u = q.front();
        q.pop();
        answer.push_back(u);

        for(auto v : adj[u]){
            if(--inDegree[v] == 0){
                q.push(v);
            }
        }

        // Vertex visited
        cnt++;
    }

    // We couldn't get a correct answer
    if (cnt != N) {
        cout << "There exists a cycle" << '\n';
        return;
    }

    // We have an answer
    for (int i = 0; i < answer.size(); i++){
        cout << answer[i] << " ";
    }
    cout << '\n';

}

/*

Time Complexity

topologicalSort -> O(V+E)

Links:

https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/
https://cp-algorithms.com/graph/topological-sort.html
https://www.geeksforgeeks.org/cpp-program-for-topological-sorting/

*/
```

2.4 Dijkstra

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

typedef long long ll;
const int N = 10; // Number of vertices

// Graph inplementation using Adjacency List ( adj[u] -> (v,w) ) (0-Based)
vector<pair<int,ll>> adj[N];

// Vector used to keep the shortest distances (0-Based)
vector<ll> dist;

// Vector used to re-construct path (0-Based)
vector<int> predecessors;

// Undirected Weighted Graph
void addEdge(int u, int v,ll w){
    adj[u].emplace_back(v,w);
    adj[v].emplace_back(u,w);
}

// Dijkstra Algorithm (Single-source shortest paths problem)
void dijkstra(int src){

    // Initialize distances from src to all other vertices as INFINITE
    dist.assign(N, LONG_LONG_MAX);

    // Initialize all predecessors to -1
    predecessors.assign(N, -1);

    //Min Heap
    priority_queue< pair<ll,int> , vector <pair<ll,int>> , greater<pair<ll,
        int>> > pq;

    // Insert source in priority queue (Weight,vertex)
    pq.push(make_pair(0, src));
    dist[src] = 0;

    while (!pq.empty()){

        // The first vertex in pq is the minimum distance vertex
        int u = pq.top().second;
        pq.pop();

        // Get all adjacent vertex of "u"
        for (auto i : adj[u]){

            // Current adjacent vertex of "u"
            int v = i.first;
            ll w = i.second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + w){

                // Updating distance of v
                dist[v] = dist[u] + w;
                pq.push(make_pair(dist[v], v));

                // Updating predecessors of v
                predecessors[v] = u;
            }
        }
    }
}

/*

Time Complexity

dijkstra -> O(E*LogV)

Links:

https://cp-algorithms.com/graph/dijkstra.html
```

```
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-
    priority_queue-stl/

*/
```

2.5 Floyd-Warshall

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

typedef long long ll;
const int N = 10; // Number of vertices

// Graph inplementation using Adjacency matrix (0-Based)
ll graph[N][N];

// Matrix used to keep the shortest distances (0-Based)
ll dist[N][N];

// Undirected Weighted Graph
void addEdge(int u, int v,ll w){
    graph[u][v] = w;
    graph[v][u] = w;
}

// Function used to initialize the distance matrix
void build(){

    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            if(i == j) dist[i][j] = 0;
            else if(graph[i][j]) dist[i][j] = graph[i][j];
            else dist[i][j] = LONG_LONG_MAX;
        }
    }

}

// Floyd-Warshall Algorithm
// Find the length of the shortest path d[i][j] between each pair of
    vertices i and j.
void floydWarshall(){

    // We test all K vertices as intermediaries between (i -> j),
    // the shortest path between (i -> j) will be (i -> k -> j)
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                // If our graph has negative weight edges it is necessary
                    to do this check
                if (graph[i][k] < __LONG_LONG_MAX__ && graph[k][j] <
                    __LONG_LONG_MAX__)
                    graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j
                        ]);
            }
        }
    }

}

/*

Time Complexity

build           -> O(N^2)
floydWarshall   -> O(N^3)

Links:

https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html

https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

Obs.:

1  ) Erros de precis o  s o acumulados muito r pido utilizando pontos
    flutuantes, precisamos corrigir
utilizando EPS (Ver o primeiro site)
2  ) Podemos guardar os predecessores utilizando uma matriz.
3  ) O grafo pode ter pesos negativos, mas  n o ciclos negativos

*/
```

2.6 Bellman-Ford

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

typedef long long ll;
const int N = 10; // Number of vertices

// Graph inplementation using Edge List (u,v,w) (0-Based)
vector<tuple<int,int,ll>> edgeList;

// Array used to keep the shortest distances (0-Based)
ll dist[N];

// Undirected Weighted Graph
void addEdge(int u, int v,ll w){
    edgeList.emplace_back(u,v,w);
    edgeList.emplace_back(v,u,w);
}

// Bellman-Ford Algorithm (Single source shortest path with negative weight
    edges)
bool bellmanFord(int src){

    // Initialize distances from src to all other vertices as INFINITE
    for(int i = 0; i < N; i++){
        dist[i] = LONG_LONG_MAX;
    }
    dist[src] = 0;

    // Relax all edges N - 1 times. If we're sure that we don't have
    // negative cycles, we can use a flag to stop when there isn't update
    for(int i = 0; i < N-1; i++){
        for(int j = 0; j < edgeList.size(); j++){
            int u,v,w;
            tie(u,v,w) = edgeList[j];
            if(dist[u] < LONG_LONG_MAX)
                dist[v] = min(dist[v], dist[u] + w);
        }

    }

    // Check for negative-weight cycles.  The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle. If we have a update, there's a negative cycle
        .
    for(int j = 0; j < edgeList.size(); j++){
        int u,v,w;
        tie(u,v,w) = edgeList[j];
        if(dist[u] < LONG_LONG_MAX && dist[u] + w < dist[v])
            return false;
    }

    // There isn't negative cycle and our answer is in dist
    return true;
}

/*
```

```
Time Complexity

bellmanFord -> O(V*E)

Links:

https://cp-algorithms.com/graph/bellman_ford.html
https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

*/
```

2.7 SPFA

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const int N = 10; // Number of vertices

// Graph inplementation using Adjacency List ( adj[u] -> (v,w) ) (0-Based)
vector<pair<int,ll>> adj[N];

// Vector used to keep the shortest distances (0-Based)
vector<ll> dist;

// Undirected Weighted Graph
void addEdge(int u, int v,ll w){
    adj[u].emplace_back(v,w);
    adj[v].emplace_back(u,w);
}

// Shortest Path Faster Algorithm (SPFA) (Single source shortest path with
    negative weight edges)
// SPFA is a improvement of the Bellman-Ford algorithm.
bool spfa(int src){

    // Initialize distances from src to all other vertices as INFINITE
    dist.assign(N, LONG_LONG_MAX);

    // Counts the number of times the distance has changed
    // (if it is greater than N-1 there is a negative cycle)
    vector<int> count(N, 0);

    // If a vertex is already in the queue, there is no need to put it back
    vector<bool> inqueue(N, false);
    queue<int> q;

    dist[src] = 0;
    q.push(src);
    inqueue[src] = true;

    // Takes advantage of the fact that not all attempts at relaxation will
        work.
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inqueue[u] = false;

        for (auto i : adj[u]) {

            int v = i.first;
            ll w = i.second;

            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (!inqueue[v]) {
                    q.push(v);
                    inqueue[v] = true;
                    count[v]++;
                    if (count[v] > N)
                        return false;  // negative cycle
                }
            }
        }
    }

    // There isn't negative cycle and our answer is in dist
    return true;
}

/*

Time Complexity

SPFA -> O(V*E) (In the worst case, on average it is O(E), so it's efficient
    )

Links:

https://cp-algorithms.com/graph/bellman_ford.html
https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

*/
```

2.8 Kruskal

```cpp
// Kruskal's algorithm uses Disjoint Set, it's necessary to include to use
    the functions "Find" and "Union"
// See in "/Data-Structures/DisjointSetUnion.cpp"

#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const int N = 10; // Number of vertex

// Structure to symbolize a weighted edge
struct Edge {
    int u, v, w;
    bool operator<(Edge const& other) {
        return w < other.w;
    }
};

// Graph inplementation using Edge List (u,v,w) (0-Based)
vector<Edge> edges;

// Vector responsible for keeping the edges used to build the MST
vector<Edge> mst;

// Kruskal's Algorithm for Minimum Spanning Tree using Disjoint Set (
    Returns the cost to build the MST)
ll kruskal(){

    ll cost = 0;

    // We can stop the algorithm when we have N-1 edges
    int cntEdges = 0;

    // Will sort edges by weight
    sort(edges.begin(), edges.end());

    for (Edge e : edges) {
        if (Find(e.u) != Find(e.v)) {
            cost += e.w;
            mst.push_back(e);
            Union(e.u, e.v);
            cntEdges++;
            if(cntEdges == N-1)
                break;
        }
```

```
    }

    return cost;
}

/*

Time Complexity

kruskal -> O(E*logN)

Disjoint Set Union Functions

Build   -> O(N)
Find    -> O(logN) ( In the worst case, the average case is O(1) )
Union   -> O(logN) ( In the worst case, the average case is O(1) )

Links:

https://cp-algorithms.com/graph/mst_kruskal.html
https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html

Obs.:

1  ) Para criar uma Maximum Spanning Tree podemos simplesmente dar um "
     reverse(edges.begin(), edges.end());"
ou caminhar pelo for no sentido contr rio "for(int i = N-1; i >= 0; i--)"

*/
```

2.9 Prim

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

typedef long long ll;
const int N = 1e5+10; // Number of vertex

// Graph inplementation using Adjacency List ( adj[u] -> (v,w) ) (0-Based)
vector<pair<int,ll>> adj[N];

// Undirected Weighted Graph
void addEdge(int u, int v,ll w){
    adj[u].emplace_back(v,w);
    adj[v].emplace_back(u,w);
}

// Prim's Algorithm for Minimum Spanning Tree (Returns the cost to build
     the MST)
ll prim(){

    //Min Heap
    priority_queue< pair<ll,int> , vector <pair<ll,int>> , greater<pair<ll,
        int>> > pq;

    // Taking vertex 0 as source
    int src = 0;

    // Cost to build the MST
    ll wMST = 0;

    // Create a vector for costs and initialize all costs as infinite and
         source as 0
    vector<ll> costs(N, LONG_LONG_MAX);
    costs[src] = 0;

    // To store parent array which in turn store MST
    vector<int> parent(N, -1);

    // To keep track of vertices included in MST
    vector<bool> inMST(N, false);

    // Insert source in priority queue (w,u)
    pq.emplace(0, src);

    while (!pq.empty()){

        // Extracting the minimum costs vertex
        int u = pq.top().second;
        ll wt = pq.top().first;
        pq.pop();

        // If the vertex has already been added there is no need to
             continue
        if(inMST[u] == true) continue;

        // Include vertex in MST (Took the shortest path possible to get to
              it)
        inMST[u] = true;
        wMST += wt;

        // Get all adjacent vertices of "u"
        for(auto i : adj[u]){
            int v = i.first;
            ll w = i.second;

            // "v" isn't in MST and weight of (u,v) is smaller than current
                 costs of "v"
            if(inMST[v] == false && costs[v] > w){
                // Updating weight to "v" (Is a possible edge that i can
                     put in the MST)
                costs[v] = w;
                pq.emplace(w,v);
                parent[v] = u;
            }
        }
    }

    /*
    // Print edges of MST using parent array (We start from 1 because the
         parent of the source is -1)
    for (int i = 1; i < N; ++i)
        printf("%d - %d\n", parent[i], i);
    */

    return wMST;
}
```

2.10 Dinic - Max Flow

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

#define int long long
using ll = long long;

const int INF = 1e18;

// Dinic's Algorithm to find the Max Flow of a graph
class Dinic{
    int N;
    vector<int> level;
    vector<bool> dead;

public:

    struct Edge{
        Edge(int vertice, int capacity){
            v = vertice;
            cap = capacity;
        }
```

```
    int v;
    int cap;
};
int source, sink;
vector<Edge> edge;
vector<vector<int>> g;

Dinic(int size, int u, int v){  // Initializing Dinic
    g.resize(size);
    N = size;
    level.resize(size);
    source = u;
    sink = v;
}

void addEdge(int u, int v, int cap){ // Making Residual Network
    g[u].push_back(edge.size());
    edge.push_back(Edge(v,cap));
    g[v].push_back(edge.size());
    edge.push_back(Edge(u,0));
}

int run(){

    int flow = 0;
    while(BFS())
        flow += maxflow(source, INF);

    return flow;
}

// Algorithm to find the edges from MinCut by dividing the graph into 2
     subgraphs.
void mincut(){

    set<int> reachable; // 2 subgraphs
    set<int> unreachable;

    vector<bool> visited(N, false);
    queue<int> q;
    q.push(source);
    reachable.insert(source);

    while(!q.empty()){  // BFS to find all vertices that are reached by
         the source using positive-capacity edges

        int u = q.front();
        visited[u] = true;
        q.pop();

        for(auto x: g[u]){
            if(!visited[edge[x].v] && edge[x].cap > 0){
                q.push(edge[x].v);
                reachable.insert(edge[x].v);
            }
        }
    }

    for(int i = 1; i < N; i++){ // If the graph is 0-based you need to
         put i = 0
        if(reachable.find(i) == reachable.end())
            unreachable.insert(i);
    }

    bool flag = true;
    for(auto i: reachable){ // Printing edges responsible for
         connecting the 2 subgraphs
        for(auto j: g[i]){
            if(unreachable.find(edge[j].v) != unreachable.end()){
                if(flag == true)cout << i << " " << edge[j].v << '\n';
                flag = flag^true;
            }
        }
        }
    }

    // Algorithm to find edges from Maximum Matching of a bipartite graph (
         may have variations which capacity need not be 0)
    void max_matching(int n, int m){

        for(int i = 1; i <= n + m; i++){
            for(auto j: g[i]){
                if(j%2 == 0 && edge[j].cap == 0){
                    if(edge[j].v != 0 && edge[j].v != n+m+1)
                        cout << i << " " << edge[j].v - n << '\n';
                }
            }
        }
    }

private:

    bool BFS(){ // Construct the Augmenting Level Path

        for(int i = 0; i < N; i++) level[i] = INF;
        dead.clear();
        dead.resize(N, false);
        level[source] = 0;
        queue<int> q;

        q.push(source);
        while(!q.empty()){

            int u = q.front();
            q.pop();

            if(u == sink) return true;

            for(auto x: g[u]){
                if(level[edge[x].v] == INF && edge[x].cap > 0){
                    level[edge[x].v] = level[u] + 1;
                    q.push(edge[x].v);
                }
            }
        }
        return false;
    }

    int maxflow(int u, int flow){

        if(dead[u] || flow == 0) return 0;
        if(u == sink) return flow;

        int answ = 0;

        for(auto i: g[u]){
            if(level[edge[i].v] != level[u] + 1) continue;
            int f = maxflow(edge[i].v, min(edge[i].cap, flow));
            int reversed_i = (i%2 == 0 ? i+1 : i-1);    // Finding the "
                 even" edge of "i"
            flow -= f;
            answ += f;
            edge[i].cap -= f;
            edge[reversed_i].cap += f;
        }

        if(answ == 0) dead[u] = true;
        return answ;
    }
};

/*

Time Complexity

Dinic                                   -> O(V E)
Dinic Maximum Matching                  -> O(E*sqrt(V))
```

```
Dinic in Unit Capacities Networks   -> O(E*V^(2/3))

Links:

https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-
    problem/
https://www.youtube.com/watch?v=oHy3ddI9X3o
https://cp-algorithms.com/graph/edmonds_karp.html
https://cp-algorithms.com/graph/dinic.html
https://www.youtube.com/watch?v=M6cm8UeeziI
https://www.youtube.com/watch?v=duKIzgJQ1w8

*/
```

2.11 Min Cost Max Flow

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

#define int long long
using ii = pair<int,int>;
using ll = long long;
const int INF = 1e18;

// Implementation of the Min Cost Flow algorithm
class MinCostFlow{
private:
    struct Edge{
        int u,v,cap,cost,flow;
        Edge(int from, int to, int cap, int cost):
            u(from), v(to), cap(cap), cost(cost), flow(0) {}
    };
    int n, source, sink;
    vector<Edge> edge;
    vector<vector<int>> g;
    vector<int> dist;
    vector<int> parent;

public:
    MinCostFlow(int n, int source, int sink){ // Initializing
        this->n = n;
        this->source = source;
        this->sink = sink;
        g.assign(n, vector<int>());
    }

    void addEdge(int u, int v, int cap, int cost){
        int m = edge.size();
        edge.emplace_back(u, v, cap, cost);
        g[u].push_back(m);
        edge.emplace_back(v, u, 0, -cost);
        g[v].push_back(m+1);
    }

    ii run(int k){  // Running Min Cost Flow, to calculate Min Cost Max
         Flow put k = INF
        ii answ = {0,0};    // {flow, cost}
        while(answ.first < k && spfa()){
            ii aux = get_flow_and_cost();   // {flow, cost}
            int max_add = min(aux.first, k - answ.first);

            answ.first += max_add;
            answ.second += max_add * aux.second;
        }
        return answ;
    }

private:
    bool spfa(){ // Shortest Path Faster Algorithm
        dist.assign(n, INF);
        parent.assign(n, -1);
        queue<int> q;
        vector<bool> inqueue(n, false);
        vector<int> count(n, 0);

        dist[source] = 0;
        q.push(source);
        inqueue[source] = true;

        while(!q.empty()){
            int u = q.front();
            q.pop();
            inqueue[u] = false;

            for(auto id: g[u]){
                Edge aux = edge[id];

                int new_dist = dist[u] + aux.cost;
                if(aux.cap - aux.flow > 0 && new_dist < dist[aux.v]){

                    parent[aux.v] = id;
                    dist[aux.v] = new_dist;

                    if(!inqueue[aux.v]){
                        q.push(aux.v);
                        inqueue[aux.v] = true;

                        count[aux.v]++;
                        if(count[aux.v] > n) return false; // Found a
                             negative cycle
                    }
                }
            }
        }
        return dist[sink] < INF;
    }

    ii get_flow_and_cost(){
        ii flow_cost = {INF,0};

        int v = sink;
        while(v != source){
            Edge aux = edge[parent[v]];
            flow_cost.first = min(flow_cost.first, aux.cap - aux.flow);
            flow_cost.second += aux.cost;
            v = aux.u;
        }

        v = sink;
        while(v != source){
            edge[parent[v]].flow += flow_cost.first;
            edge[parent[v] ^ 1].flow -= flow_cost.first;
            v = edge[parent[v]].u;
        }
        return flow_cost;
    }
};

/*

Time Complexity

MinCostFlow -> O(n^2*m^2)

Links:

https://cp-algorithms.com/graph/min_cost_flow.html#algorithm
https://cp-algorithms.com/graph/bellman_ford.html#shortest-path-faster-
    algorithm-spfa
https://www.youtube.com/watch?v=AtkEpr7dsW4
https://blog.thomasjungblut.com/graph/mincut/mincut/

*/
```

2.12 Stoer Wagner Min Cut

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

#define int long long
using ll = long long;

// Graph implementation using matrix
vector<vector<int>> g;

// Implementation of the Stoer-Wagner algorithm to find the global min cut
     of a graph
// We'll use this algorithm for problems where we don't know exactly where
     the source or sink is
int stoer_wagner(int n){
    int answ = LONG_LONG_MAX;
    for(int i = 1; i < n; i++){   // Need to repeat only n-1 times
        vector<int> weight(n, 0);
        vector<bool> visited(n, false);

        int prev = -1, v = 0, cnt = 1, current_cut = 0;
        while(cnt <= n-i){ // Creating the subset until only 1 node is
             missing
            visited[v] = true;
            int nxt = -1;
            current_cut = 0;

            for(int j = 0; j < n; j++){ // Looking for edge with the
                 greatest weight connected to current subset
                weight[j] += g[v][j];
                if(!visited[j] && weight[j] > current_cut){
                    nxt = j;
                    current_cut = weight[j];
                }
            }

            cnt++;
            prev = v;
            v = nxt;
        } // At the end of the loop "v" is the disconnected node and "prev"
             was the last node that reached it

        answ = min(answ, current_cut);
        for(int j = 0; j < n; j++){ // Joining the last 2 nodes, putting
             the edges of "v" in "prev"
            if(j != prev){
                g[j][prev] += g[j][v];
                g[prev][j] += g[v][j];
            }
            g[j][v] = 0; // Emptying the weights to "v" because "v" will no
                 longer exist
        }
    }
    return answ;
}


/*

Time Complexity

stoer_wagner    -> O(V^3) can be optimized to O(|V|*|E| + |V|  *log|V|)

Links:

https://www.youtube.com/watch?v=AtkEpr7dsW4
https://blog.thomasjungblut.com/graph/mincut/mincut/

*/
```

2.13 LCA (Binary Lifting)

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

typedef long long ll;

const int MAXN = 10000;          // Number of vertices
const int LOG = 20;              // Log2(N)

vector<vector<int>> children;   // Graph inplementation using Adjacency
     List (0-Based)
int up[MAXN][LOG];               // up[v][j] is 2^j-th ancestor of v
int depth[MAXN];

// Preprocessing
void dfs(int a, int p) {
        for(int b : children[a]) {
                if(b == p) continue;    // don't go back to the father
                depth[b] = depth[a] + 1;
                up[b][0] = a;                   // a is parent of b
                for(int j = 1; j < LOG; j++) {
                        up[b][j] = up[ up[b][j-1] ][j-1];
                }
                dfs(b, a);
        }
}

// Algorithm to find the Lowest Common Ancestor using Binary Lifting
int lca(int a, int b) {
        if(depth[a] < depth[b]) {
                swap(a, b);
        }
        // 1) Get same depth.
        int k = depth[a] - depth[b];
        for(int j = LOG - 1; j >= 0; j--) {
                if(k & (1 << j)) {
                        a = up[a][j]; // parent of a
                }
        }
        // 2) if b was ancestor of a then now a == b
        if(a == b) {
                return a;
        }
        // 3) move both a and b with powers of two
        for(int j = LOG - 1; j >= 0; j--) {
                if(up[a][j] != up[b][j]) {
                        a = up[a][j];
                        b = up[b][j];
                }
        }
        return up[a][0];
}

/*

Time Complexity

lca     -> O(logN)
dfs     -> O(N*logN)

Links:

https://cp-algorithms.com/graph/lca_binary_lifting.html
https://www.youtube.com/watch?v=dOAxrhAUIhA
https://github.com/Errichto/youtube/blob/master/lca.cpp

*/
```

2.14 LCA (Segment Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

const int N = 2000; // Number of vertices

// Graph inplementation using Adjacency List (0-Based)
vector<int> adj[N];

// Implementation of a Recursive Segment Tree (1-Based) {Node,Height} (Min Operation)
vector<pair<ll,ll>> segTree;

// Vector to save the Euler tour (1-Based)
vector<ll> euler;

// Auxiliary vectors
vector<ll> height, first;
vector<bool> visited;

// Call using build(1,1,N) (N = v.size() - 1)
void build_SegTree(int p, int l, int r){

    // Building Leaf
    if(l == r){
        segTree[p] = {euler[r], height[euler[r]]};
    }
    else{

        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        // Building Left Children
        build_SegTree(lc, l, m);

        // Building Rigth Children
        build_SegTree(rc, m+1, r);

        // Building Node (Min Operation)
        segTree[p] = (segTree[lc].second < segTree[rc].second) ? segTree[lc] : segTree[rc];
    }
}

// Call using query(1,1,N,ql,qr) (ql and qr need to be 1-Based)
pair<ll,ll> query_SegTree(int p, int l, int r, int ql, int qr){

    // This node is inside the range answer
    if(ql <= l && r <= qr){
        return segTree[p];
    }
    else{

        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;

        // Our answer is just in the Left Children
        if(qr <= m){

            return query_SegTree(lc,l,m,ql,qr);
        }
        // Our answer is just in the Rigth Children
        else if(ql > m){

            return query_SegTree(rc,m+1,r,ql,qr);
        }
        else{

            // Our answer is an intersection of the 2 sides
            return (query_SegTree(lc,l,m,ql,qr).second < query_SegTree(rc,m+1,r,ql,qr).second) ? query_SegTree(lc,l,m,ql,qr) : query_SegTree(rc,m+1,r,ql,qr);
        }
    }
}

// Make the Euler Tour of the tree
void dfs(ll node, ll h){

    visited[node] = true;
    height[node] = h;
    first[node] = euler.size();

    euler.push_back(node);

    for(auto u : adj[node]){
        if(!visited[u]){
            dfs(u, h+1);
            euler.push_back(node);  // We need to put it on when we come back
        }
    }
}

// Function to build the SegTree to be used in the LCA
void build_LCA(ll root){

    height.resize(N);   // Setting up
    height.push_back(-1);
    first.resize(N);

    euler.reserve(N * 2);
    euler.push_back(-1);    // 1-Based

    visited.assign(N, false);

    dfs(root, 0);   // Making Euler Tour of the Tree

    ll m = euler.size();   // Making Segment Tree
    segTree.resize(m * 4);
    build_SegTree(1, 1, m - 1);
}

// Algorithm to find the Lowest Common Ancestor using a Segmentation Tree
ll lca(int u,int v){

    int l = first[u];
    int r = first[v];

    if(l > r){
        swap(l,r);
    }

    return query_SegTree(1, 1, euler.size()-1, l, r).first;
}

/*

Time Complexity

lca         -> O(logN)
build_LCA   -> O(N)

Links:

https://cp-algorithms.com/graph/lca.html

*/
```

2.15 Kosaraju (Strongly Connected Components)

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const int N = 9; // Number of vertex

// Graph inplementation using Adjacency List (0-Based)
vector<int> adj[N];

// Transposed Graph using Adjacency List (0-Based)
vector<int> adjT[N];

// Condensation Graph using Adjacency List (0-Based)
vector<int> adjCond[N];

// Auxiliary Vectors
vector<bool> visited;
vector<int> order;
vector<int> component;

// Nodes of the same component will have the same root
vector<int> roots;

// List of all root nodes (one per component) in the Condensation Graph.
vector<int> root_nodes;

// Directed Graph (Building the graph and its transposition)
void addEdge(int u, int v){
    adj[u].push_back(v);
    adjT[v].push_back(u);
}

// First DFS traversal to build the processing order
void dfsOrder(int u){

    visited[u] = true;

    for(auto v : adj[u]){
        if(!visited[v])
            dfsOrder(v);
    }

    order.push_back(u);
}

// Second DFS Traversal to build the list of components
void dfsComponent(int u) {
    visited[u] = true;
    component.push_back(u);

    for (auto v : adjT[u])
        if (!visited[v])
            dfsComponent(v);
}

// Kosaraju's Algorithm for finding Strongly Connected Components (SCC)
void kosaraju(){

    // Building the processing order
    visited.assign(N,false);

    for(int i = 0; i < N; i++)
        if (!visited[i])
            dfsOrder(i);

    // We need to reverse the order
    visited.assign(N, false);
    reverse(order.begin(), order.end());
    roots.assign(N, 0);

    // Finding Strongly Connected Components Through Graph Transpose
    for (auto u : order){
        if (!visited[u]) {
            dfsComponent(u);

            // We put the first one on the list as the component's
                representative
            int root = component.front();
            for (auto v : component)
                roots[v] = root;

            // This node exists in the Condensed Graph
            root_nodes.push_back(root);

            component.clear();
        }
    }

    // Building the Condensed Graph
    for (int u = 0; u < N; u++){
        for (auto v : adj[u]) {
            int root_u = roots[u];
            int root_v = roots[v];

            if (root_u != root_v)
                adjCond[root_u].push_back(root_v);
        }
    }
}

// Testing the Algorithm
int main(){

    addEdge(0,1);
    addEdge(1,2);
    addEdge(2,3);
    addEdge(3,0);
    addEdge(2,4);
    addEdge(4,5);
    addEdge(5,6);
    addEdge(6,4);
    addEdge(7,6);
    addEdge(7,8);

    kosaraju();

    cout << "Nos existentes no grafo condensado: " << '\n';
    for(int i = 0; i < root_nodes.size(); i++){
        cout << root_nodes[i] << " ";
    }
    cout << "\n";

    cout << "Grafo Condensado: " << "\n";
    for(int i = 0; i < N; i++){
        cout << i << " -> ";
        for(auto u : adjCond[i]){
            cout << u << " ";
        }
        cout << '\n';
    }
    return 0;
}

/*

Time Complexity

kosaraju    -> O(V+E)

Links:
```

```
https://cp-algorithms.com/graph/strongly-connected-components.html
https://www.geeksforgeeks.org/strongly-connected-components/
https://www.youtube.com/watch?v=5wFyZJ8yH9Q

*/
```

2.16 2 SAT

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

typedef long long ll;
const int N = 5;     // Number of variables

// Graph inplementation using Adjacency List (1-Based)
vector<int> adj[2*N+1];

// Transposed Graph using Adjacency List (1-Based)
vector<int> adjT[2*N+1];

// Auxiliary Vectors
vector<int> visited;
vector<int> order;
vector<int> comp;

// We set a variable to true or false (if there's an answer)
vector<bool> assignment;

// Directed Graph (Building the graph and its transposition)
void addEdge(int u, int v){

    // Variable x is mapped to x
    // Variable -x is mapped to N+x = N-(-x)

    // For (x v y) we need to add edges (!x -> y) and (!y -> x)
    if(u > 0 && v > 0){

        adj[u+N].push_back(v); // (!x -> y)
        adjT[v].push_back(u+N);

        adj[v+N].push_back(u); // (!y -> x)
        adjT[u].push_back(v+N);
    }

    else if(u > 0 && v < 0){

        adj[u+N].push_back(N-v); // (!x -> y)
        adjT[N-v].push_back(u+N);

        adj[-v].push_back(u);  // (!y -> x)
        adjT[u].push_back(-v);
    }

    else if (u < 0 && v > 0){

        adj[-u].push_back(v); // (!x -> y)
        adjT[v].push_back(-u);

        adj[v+N].push_back(N-u);  // (!y -> x)
        adjT[N-u].push_back(v+N);
    }

    else{

        adj[-u].push_back(N-v); // (!x -> y)
        adjT[N-v].push_back(-u);

        adj[-v].push_back(N-u);  // (!y -> x)
        adjT[N-u].push_back(-v);
    }
}

// First DFS traversal to build the processing order
void dfsOrder(int u){
    visited[u] = true;

    for(auto v : adj[u]){
        if(!visited[v])
            dfsOrder(v);
    }

    order.push_back(u);
}

// Second DFS Traversal to build the list of components
void dfsComponent(int u, int cl){
    comp[u] = cl;

    for (auto v : adjT[u])
        if (comp[v] == -1)
            dfsComponent(v, cl);
}

// 2SAT solution based on Kosaraju's Algorithm for finding Strongly
      Connected Components (SCC)
bool solve2SAT(){

    // Building the processing order
    visited.assign(2*N+1,0);

    for(int i = 1; i <= 2*N; i++) // (1-Based)
        if (!visited[i])
            dfsOrder(i);

    // We need to reverse the order
    reverse(order.begin(), order.end());

    comp.assign(2*N+1, -1);
    for (int i = 0, j = 0; i < 2*N; i++) {
        int u = order[i];
        if (comp[u] == -1)
            dfsComponent(u, j++);
    }

    // Building our answer
    assignment.assign(N+1, false);

    for (int i = 1; i <= N; i++) { // (1-Based)
        if (comp[i] == comp[i+N])
            return false;
        assignment[i] = comp[i] > comp[i+N];
    }
    return true;
}

// Testing the Algorithm
int main(){

    // (x1 v x2)^(!x2 v x3)^(!x1 v !x2)^(x3 v x4)^(!x3 v x5)^(!x4 v !x5)^(!
       x3 v x4)
    int a[] = {1, -2, -1, 3, -3, -4, -3};
    int b[] = {2, 3, -2, 4, 5, -5, 4};

    for(int i = 0; i < 7; i++){
        addEdge(a[i],b[i]);
    }

    if(solve2SAT()){
        cout << "YES" << '\n';
        cout << "Possible Answer: ";
        for(int i = 1; i <= N; i++){
            cout << assignment[i] << " ";
        }
```

```
        cout << '\n';
    }
    else{
        cout << "NO" << '\n';
    }

    return 0;
}

/*

Time Complexity

solve2SAT   -> O(V+E)

Links:

https://cp-algorithms.com/graph/2SAT.html
https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/

Obs.:

1  ) A entrada das variaveis precisam estar na Conjunctive Normal Form (CNF
    ) para o algoritmo funcionar
2  ) Lembre-se que: (A v B) = true,    igual a ===> (!A -> B) AND (!B -> A)
    , Com isso transformamos (A V B) em 2 arestas.

*/
```

3 Dynamic Programming

3.1 LCS

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

#define int long long
const int MAXN = 1e4;
int dp[MAXN][MAXN];
string s1, s2;

// Longest Common Subsequence
// Call lcs(0, 0)
int lcs(int i, int j){
    if(i == s1.size() || j == s2.size()) return 0;

    if(dp[i][j] != -1) return dp[i][j];

    if(s1[i] == s2[j])
        return dp[i][j] = 1 + lcs(i + 1, j + 1);

    return dp[i][j] = max(lcs(i + 1, j), lcs(i, j + 1));
}

string answ;

// Recovery lcs answer
void recovery(int i, int j){
    if(i == s1.size() || j == s2.size()) return;
    if(s1[i] == s2[j] && dp[i][j] == 1 + lcs(i + 1, j + 1)){
        answ.push_back(s1[i]);
        recovery(i + 1, j + 1);
    }
    else if(dp[i][j] == lcs(i + 1, j)) recovery(i + 1, j);
    else recovery(i, j + 1);
}

/*

Time Complexity

lcs -> O(n*m)

Links:

https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/
https://www.youtube.com/watch?v=sSno9rV8Rhg

*/
```

3.2 LIS

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

#define int long long

vector<int> v;

// Longest Increasing Subsequence
int lis(int n){
    vector<int> dp(n, 1);
    for(int i = 0; i < n; i++){
        for(int j = 0; j < i; j++){
            if(v[j] < v[i]) dp[i] = max(dp[i], dp[j] + 1);
        }
    }
    int answ = 0;
    for(int i = 0; i < n; i++) answ = max(answ, dp[i]);
    return answ;
}

// Longest Increasing Subsequence using Binary Search
vector<int> LISBS(int n){
    const int INF = 0x3f3f3f3f3f3f3f3f;
    vector<int> dp(n + 1, INF);
    vector<int> idx(n + 1, -1);
    vector<int> parent(n + 1, -1);
    dp[0] = -INF;

    for(int i = 0; i < n; i++){
        int j = upper_bound(dp.begin(), dp.end(), v[i]) - dp.begin();
        if(dp[j-1] < v[i] && v[i] < dp[j]){
            dp[j] = v[i];
            idx[j] = i;
            parent[j] = idx[j-1];
        }
    }

    vector<int> answ;
    int pos = 0;
    for(int i = 0; i <= n; i++){
        if(dp[i] < INF) pos = i;
    }
    while(pos != -1){
        answ.push_back(v[pos]);
        pos = parent[pos];
    }
    return answ;
}

/*

Time Complexity

lis     -> O(n^2)
LISBS   -> O(n*logn)

Links:
```

```
https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/
https://www.youtube.com/watch?v=1RpMc3fv0y4

*/
```

4 Number Theory

4.1 Fast Exponentiation

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ll = long long;

// Fast Modular Exponentiation
int fastModExp(int x, int y, int p){
    int ans = 1;
    x = x % p;      // Update 'x' if it is more than or equal to 'p'
    if (x == 0) return 0;   // In case 'x' is divisible by 'p';
    while(y){ // We walk through the bits of power "y"
        if (y & 1) ans = (ans*x) % p; // If the least significant bit is
             set, we multiply the answer by "x"
        y = y >> 1; // We walk to the next bit
        x = (x*x) % p;
    }
    return ans;
}

/*

Time Complexity

fastModExp  -> O(logy)

Links:

https://www.youtube.com/watch?v=HN7ey_-A7o4
https://www.youtube.com/watch?v=-3Lt-EwR_Hw
https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-
     arithmetic/

*/
```

4.2 Multiplicative Inverse

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ll = long long;

// If we know MOD is prime, then we can use Fermats s little theorem to
     find the inverse.
int ModMultInv(int n){
    return fastModExp(n,MOD-2,MOD)%MOD;
}

/*

Time Complexity

ModMultInv -> O(logMOD)

Links:

https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/

*/
```

4.3 Miller Rabin - Pollard Rho

```cpp
#include <bits/stdc++.h>
using namespace std;

// This code becomes inefficient for numbers greater to 10^20
// To numbers greater than 2^64 use __int128
typedef unsigned long long ull;

// Miller Rabin

ull mul(ull a, ull b, ull mod){
    ull answ = 0;
    for(a %= mod, b %= mod; b != 0;
        b >>= 1, a <<= 1, a = a >= mod ? a - mod : a){
        if(b & 1){
            answ += a;
            if(answ >= mod) answ -= mod;
        }
    }
    return answ;
}

ull mpow(ull a, ull b, ull mod){
    ull answ = 1;
    for(; b; b >>= 1, a = mul(a, a, mod))
        if(b & 1) answ = mul(answ, a, mod);
    return answ;
}

bool witness(ull a, ull k, ull q, ull n){
    ull t = mpow(a, q, n);
    if(t == 1 || t == n-1) return false;
    for(int i = 0; i < k - 1; i++){
        t = mul(t, t, n);
        if(t == 1) return true;
        if(t == n - 1) return false;
    }
    return true;
}

vector<ull> test = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
bool isPrime(ull n){
    if(n == 2) return true;
    if(n < 2 || !(n & 1)) return false;
    ull q = n - 1, k = 0;
    while(!(q & 1)) q >>= 1, k++;
    for(ull a : test){ // Maybe larger numbers than 1e9 is enough to
         generate 2 numbers rand()%(n-4) + 2;
        if(n == a) return true;
        if(witness(a, k, q, n)) return false;
    }
    return true;
}

// Pollard Rho

ull pollard_rho(ull n, ull c){
    ull x = 2, y = 2, i = 1, k = 2, d;
    while(1){
        x = (mul(x, x, n) + c);
        if(x >= n) x -= n;
        d = __gcd(x - y, n);
        if(d > 1) return d;
        if(++i == k) y = x, k <<= 1;
    }
    return n;
}
```

```cpp
void factorize(vector<ull>& answ, ull n){
    if(n == 1) return;
    if(isPrime(n)){
        answ.push_back(n);
        return;
    }
    ull d = n;
    for(int i = 2; d == n; i++) d = pollard_rho(n, i);
    factorize(answ, d);
    factorize(answ, n/d);
}

// cin/cout to deal with __int128

istream& operator>>(istream& in, ull &x){
    static char s[40];
    in >> s;
    x = 0;
    for(char* p = s; *p; ++p) x = 10 * x + *p - '0';
    return in;
}

ostream& operator<<(ostream& out, ull x){
    static char s[40] = {};
    char* p = s + (sizeof(s) - 1);
    while (*--p = (char)(x % 10 + '0'), x /= 10, x);
    return out << p;
}

/*

Time Complexity

Miller_Rabin    -> O(K*log^3(N)) Where N is the number to be checked for
    primality, and K is the number of checks to get accuracy
Pollard_Rho     -> O(n^1/4)

Links:

Miller Rabin

https://cp-algorithms.com/algebra/primality_tests.html#deterministic-
    version
https://www.geeksforgeeks.org/multiply-large-integers-under-large-modulo/
https://www.youtube.com/watch?v=qdylJqXCDGs
https://www.youtube.com/watch?v=zmhUlVck3J0

Pollard Rho

https://cp-algorithms.com/algebra/factorization.html
https://www.youtube.com/watch?v=6khEMeU8Fck

*/
```

4.4 Miller - Rho Iterative

```cpp
#include <bits/stdc++.h>
using namespace std;

// This code becomes inefficient for numbers greater to 10^20
// To numbers greater than 2^64 use __int128 and maybe __float128
typedef unsigned long long ull;
typedef long double ld;
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

// Miller Rabin Primality Test

// Just a multiplication to avoid overflow
ull fmul(ull a, ull b, ull m){
    ull q = (ld) a * (ld) b / (ld) m;
    ull r = a * b - q * m;
    return (r + m) % m;
}

// Fast Modular Exponentiation
ull fexp(ull x, ull y, ull m){
    ull answ = 1;
    x = x % m;
    while(y){
        if(y & 1) answ = fmul(answ, x, m);
        x = fmul(x, x, m);
        y = y >> 1;
    }
    return answ;
}

// Validation by F e r m a t s Small Theorem
// a^(p-1) - 1 = 0 mod p
// (a^((p-1)/2) - 1)*(a^((p-1)/2) + 1) = 0 mod p
bool miller(ull p, ull a){
    ull s = p - 1;
    while(s % 2 == 0) s >>= 1;
    while(a >= p) a >>= 1;
    ull mod = fexp(a, s, p);
    while(s != p - 1 && mod != 1 && mod != p-1){
        mod = fmul(mod, mod, p);
        s <<= 1;
    }
    if(mod != p - 1 && s % 2 == 0) return false;
    else return true;
}

// Deterministic Miller Rabin algorithm
// We need to check for different bases "a" to increase the probability of
    hit
// For values greater than 2^64 add more bases
vector<ull> test = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
bool prime(ull p){
    if(p < 2) return false;
    if(p % 2 == 0) return (p == 2);
    for(ull a : test){
        if(p == a) return true;
        if(!miller(p, a)) return false;
    }
    return true;
}

// Pollard Rho

// Function used in Pollard Rho f(x) = x^2 + c
ull func(ull x, ull c, ull n){
    return (fmul(x, x, n) + c) % n;
}

ull gcd(ull a, ull b){
    if(!b) return a;
    else return gcd(b, a % b);
}

// Pollard Rho algorithm to discover a factor of n
ull rho(ull n){
    if(n % 2 == 0) return 2;
    if(prime(n)) return n;
    while(1){
        ull c;
        do {
            c = uniform_int_distribution<ull>(0, n - 1)(rng);
        } while(c == 0 || (c + 2) % n == 0);
        ull x = 2, y = 2, d = 1;
        ull pot = 1, lam = 1;
        do {
            if(pot == lam){
                x = y;
                pot <<= 1;
```

```
                lam = 0;
            }
            y = func(y, c, n);
            lam++;
            d = gcd(x >= y ? x - y : y - x, n);
        } while(d == 1);
        if(d != n) return d;
    }
}

// Return all the factors of n
vector<ull> factors(ull n){
    vector<ull> answ, rest, times;
    if(n == 1) return answ;
    rest.push_back(n);
    times.push_back(1);
    while(!rest.empty()){
        ull x = rho(rest.back());
        if(x == rest.back()){
            int freq = 0;
            for(int i = 0; i < rest.size(); i++){
                int cur_freq = 0;
                while(rest[i] % x == 0){
                    rest[i] /= x;
                    cur_freq++;
                }
                freq += cur_freq * times[i];
                if(rest[i] == 1){
                    swap(rest[i], rest.back());
                    swap(times[i], times.back());
                    rest.pop_back();
                    times.pop_back();
                    i--;
                }
            }
            while(freq--){
                answ.push_back(x);
            }
            continue;
        }
        ull e = 0;
        while(rest.back() % x == 0){
            rest.back() /= x;
            e++;
        }
        e *= times.back();
        if(rest.back() == 1){
            rest.pop_back();
                        times.pop_back();
        }
        rest.push_back(x);
                times.push_back(e);
    }
    return answ;
}

/*

Time Complexity

Miller_Rabin    -> O(K*log^3(N)) Where N is the number to be checked for
    primality, and K is the number of checks to get accuracy
Pollard_Rho     -> O(n^1/4)

Links:

Miller Rabin

https://cp-algorithms.com/algebra/primality_tests.html#deterministic-
    version
https://www.geeksforgeeks.org/multiply-large-integers-under-large-modulo/
https://www.youtube.com/watch?v=qdylJqXCDGs
https://www.youtube.com/watch?v=zmhUlVck3J0

Pollard Rho

https://cp-algorithms.com/algebra/factorization.html
https://www.youtube.com/watch?v=6khEMeU8Fck

*/
```

4.5 Fast Fourier transform

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

typedef double ld;
const ld PI = acos(-1);

struct Complex {
    ld real, imag;
    Complex conj() { return Complex(real, -imag); }
    Complex(ld a = 0, ld b = 0) : real(a), imag(b) {}
    Complex operator + (const Complex &o) const { return Complex(real + o.
        real, imag + o.imag); }
    Complex operator - (const Complex &o) const { return Complex(real - o.
        real, imag - o.imag); }
    Complex operator * (const Complex &o) const { return Complex(real * o.
        real - imag * o.imag, real * o.imag + imag * o.real); }
    Complex operator / (ld o) const { return Complex(real / o, imag / o); }
    void operator *= (Complex o) { *this = *this * o; }
    void operator /= (ld o) { real /= o, imag /= o; }
};

typedef vector<Complex> CVector;
const int ms = 1 << 22;
int bits[ms];
Complex root[ms];

// Start by calling this function
void initFFT() {
    root[1] = Complex(1);
    for(int len = 2; len < ms; len += len) {
        Complex z(cos(PI / len), sin(PI / len));
        for(int i = len / 2; i < len; i++) {
            root[2 * i] = root[i];
            root[2 * i + 1] = root[i] * z;
        }
    }
}

void pre(int n) {
    int LOG = 0;
    while(1 << (LOG + 1) < n) {
        LOG++;
    }
    for(int i = 1; i < n; i++) {
        bits[i] = (bits[i >> 1] >> 1) | ((i & 1) << LOG);
    }
}

CVector fft(CVector a, bool inv = false) {
    int n = a.size();
    pre(n);
    if(inv) {
        reverse(a.begin() + 1, a.end());
    }
    for(int i = 0; i < n; i++) {
        int to = bits[i];
        if(to > i) {
            swap(a[to], a[i]);
        }
    }
```

```
    for(int len = 1; len < n; len *= 2) {
        for(int i = 0; i < n; i += 2 * len) {
            for(int j = 0; j < len; j++) {
                Complex u = a[i + j], v = a[i + j + len] * root[len + j];
                a[i + j] = u + v;
                a[i + j + len] = u - v;
            }
        }
    }
    if(inv) {
        for(int i = 0; i < n; i++)
            a[i] /= n;
    }
    return a;
}

// NTT
void fft2in1(CVector &a, CVector &b) {
    int n = (int) a.size();
    for(int i = 0; i < n; i++) {
        a[i] = Complex(a[i].real, b[i].real);
    }
    auto c = fft(a);
    for(int i = 0; i < n; i++) {
        a[i] = (c[i] + c[(n-i) % n].conj()) * Complex(0.5, 0);
        b[i] = (c[i] - c[(n-i) % n].conj()) * Complex(0, -0.5);
    }
}

// NTT
void ifft2in1(CVector &a, CVector &b) {
    int n = (int) a.size();
    for(int i = 0; i < n; i++) a[i] = a[i] + b[i] * Complex(0, 1);
    a = fft(a, true);
    for(int i = 0; i < n; i++) {
        b[i] = Complex(a[i].imag, 0);
        a[i] = Complex(a[i].real, 0);
    }
}

// NTT
vector<long long> mod_mul(const vector<long long> &a, const vector<long
     long> &b, long long cut = 1 << 15) {
    int n = (int) a.size();
    CVector C[4];
    for(int i = 0; i < 4; i++) C[i].resize(n);
    for(int i = 0; i < n; i++) {
        C[0][i] = a[i] % cut;
        C[1][i] = a[i] / cut;
        C[2][i] = b[i] % cut;
        C[3][i] = b[i] / cut;
    }
    fft2in1(C[0], C[1]);
    fft2in1(C[2], C[3]);
    for(int i = 0; i < n; i++) {
        // 00, 01, 10, 11
        Complex cur[4];
        for(int j = 0; j < 4; j++) cur[j] = C[j/2+2][i] * C[j % 2][i];
        for(int j = 0; j < 4; j++) C[j][i] = cur[j];
    }
    ifft2in1(C[0], C[1]);
    ifft2in1(C[2], C[3]);
    vector<long long> ans(n, 0);
    for(int i = 0; i < n; i++) {
        // if there are negative values, care with rounding
        ans[i] += (long long) (C[0][i].real + 0.5);
        ans[i] += (long long) (C[1][i].real + C[2][i].real + 0.5) * cut;
        ans[i] += (long long) (C[3][i].real + 0.5) * cut * cut;
    }
    return ans;
}

// Function to perform the multiplication of polynomials
vector<int> mul(const vector<int> &a, const vector<int> &b) {
    int n = 1;
    while (n - 1 < (int) a.size() + (int) b.size() - 2) n += n;
    CVector poly(n);
    for(int i = 0; i < n; i++) {
        if(i < (int) a.size()) {
            poly[i].real = a[i];
        }
        if(i < (int) b.size()) {
            poly[i].imag = b[i];
        }
    }
    poly = fft(poly);
    for(int i = 0; i < n; i++) {
        poly[i] *= poly[i];
    }
    poly = fft(poly, true);
    vector<int> c(n, 0);
    for(int i = 0; i < n; i++) {
        c[i] = (int) (poly[i].imag / 2 + 0.5);
    }
    while (c.size() > 0 && c.back() == 0) c.pop_back();
    return c;
}

/*

Time Complexity

fft  -> O(N*logN)

Links:

https://unacademy.com/class/fft-and-convulutions/AFIJV6BI
https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-
     multiplication/
https://cp-algorithms.com/algebra/fft.html

Applications:

1. All possible sums
2. All possible scalar products
3. Two stripes
4. String matching
5. String matching with wildcards

*/
```

4.6 Pollard Rho with Montgomery

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

// Algorithm for factoring numbers up to 10^29
// I d o n  t know how it works yet

typedef unsigned long long u64;
typedef __int128_t i128;
typedef __uint128_t u128;

struct u256 {
  u128 hi, lo;
  static u256 mult(u128 x, u128 y) {
    u128 a = x >> 64, b = (u64)x;
    u128 c = y >> 64, d = (u64)y;
    u128 ac = a * c;
    u128 ad = a * d;
    u128 bc = b * c;
    u128 bd = b * d;
```

```
    u128 carry = (u128)(u64)ad + (u64)bc + (bd >> 64u);
    u128 h = ac + (ad >> 64u) + (bc >> 64u) + (carry >> 64u);
    u128 l = (ad << 64u) + (bc << 64u) + bd;
    return {h, l};
  }
};

struct Montgomery {
  u128 n, inv, r2;
  explicit Montgomery(u128 _n) : n(_n), inv(1), r2(-n % n) {
    assert(n & 1);
    for (int i = 0; i < 7; ++i) inv *= 2 - n * inv;
    for (int i = 0; i < 4; ++i) if ((r2 <<= 1) >= n) r2 -= n;
    for (int i = 0; i < 5; ++i) r2 = mult(r2, r2);
  }
  u128 init(u128 x) { return mult(x, r2); }
  u128 mult(u128 a, u128 b) { return reduce(u256::mult(a, b)); }
  u128 reduce(u256 x) {
    i128 a = x.hi - u256::mult(x.lo * inv, n).hi;
    return (a < 0) ? a + n : a;
  }
};

istream& operator>>(istream& in, u128 &x) {
  static char s[40];
  in >> s;
  x = 0;
  for (char* p = s; *p; ++p) x = 10 * x + *p - '0';
  return in;
}

ostream& operator<<(ostream& out, u128 x) {
  static char s[40] = {};
  char* p = s + (sizeof(s) - 1);
  while (*--p = (char)(x % 10 + '0'), x /= 10, x);
  return out << p;
}

#define rand() uid(rng)
mt19937 rng(chrono::high_resolution_clock::now().time_since_epoch().count()
    );
uniform_int_distribution<int> uid(0, numeric_limits<int>::max());

inline u128 gcd(u128 a, u128 b) {
  if (b != 0) while ((b ^= a ^= b ^= a %= b));
  return a;
}

inline u128 add(u128 a, u128 b, u128 m) { return (a += b) >= m ? a - m : a;
    }
inline u128 sub(u128 a, u128 b, u128 m) { return a < b ? a + m - b : a - b;
    }

u128 mult(u128 a, u128 b, u128 m) {
  u128 x = 0;
  while (b) {
    if (b & 1) x = add(x, a, m);
    b >>= 1;
    a = add(a, a, m);
  }
  return x;
}

u128 mpow(u128 a, u128 b, u128 mod) {
  u128 x = 1;
  while (b) {
    if (b & 1) x = mult(x, a, mod);
    b >>= 1;
    a = mult(a, a, mod);
  }
  return x;
}

u128 isqrt(u128 n) {
  u128 x = n, y = 1;
  while (x > y) {
    x = (x + y) >> 1;
    y = n / x;
  }
  return x;
}

bool isPrime(u128 n) {
  if (n < 2) return 0;
  if ((n & 1) == 0) return n == 2;
  u128 d = n - 1;
  int r = 0;
  while ((d & 1) == 0) {
    d >>= 1;
    ++r;
  }
  for (u128 a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) if ((a
      %= n)) {
    a = mpow(a, d, n);
    if (a == 1 || a == n - 1) continue;
    for (int i = 1; i < r && 1 < a && a != n - 1; ++i) a = mult(a, a, n);
    if (a != n - 1) return 0;
  }
  return 1;
}

u128 rho(u128 n) {
  if (isPrime(n)) return 1;
  if ((n & 1) == 0) return 2;
  u128 r = isqrt(n);
  if (r * r == n) return r;
  Montgomery mont(n);
  const int m = n < (1ull << 60) ? 32 : 512;
  const auto one = mont.init(1);
  const auto c = mont.init((rand() & 1023) + 1);
  auto f = [&](u128 x){ return add(mont.mult(x, x), c, n); };
  u128 x = 0, y = one;
  for (int l = 1; ; l <<= 1) {
    if (x == y) y = mont.init(rand());
    x = y;
    for (int i = 0; i < l; ++i) y = f(y);
    u128 ys = y, q = one, g = 1;
    for (int k = 0; k < l && g == 1; k += m) {
      ys = y;
      for (int i = min(m, l - k); i; --i) {
        y = f(y);
        q = mont.mult(q, sub(y, x, n));
      }
      g = gcd(n, q);
    }
    if (g == n) {
      y = ys;
      for (g = 1; g == 1; g = gcd(n, sub(y, x, n))) y = f(y);
    }
    if (g != 1 && g != n) return g;
  }
}

map<u128,int> findFactors(u128 n) {
  assert(n > 1);
  u128 x = rho(n);
  if (x == 1) return {{n, 1}};
  auto a = findFactors(x);
  auto b = findFactors(n / x);
  if (a.size() < b.size()) swap(a, b);
  for (auto i : b) a[i.first] += i.second;
  return a;
}

signed main() {
  // assert(freopen("in", "r", stdin));
  cin.sync_with_stdio(0), cin.tie(0), cout.tie(0);
  u128 n;
```

```
    while (cin >> n && n) {
      bool flag = 0;
      map<u128,int> aux = findFactors(n);
      for (auto i : aux) {
        if (flag) cout << ' ';
        flag = 1;
        cout << i.first << '^' << i.second;
      }
      cout << '\n';
    }
    cerr << (double)clock() / CLOCKS_PER_SEC << endl;
    return 0;
}
```

5 Geometry

5.1 Geometry

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

const double inf = 1e100, eps = 1e-12;
const double PI = acos(-1.0L);

int cmp(double a, double b = 0){
    if(abs(a-b) < eps) return 0;
    return (a < b) ? -1 : +1;
}

// Struct to represent a point/vector
struct PT{
    double x, y;
    PT(double x = 0, double y = 0) : x(x), y(y) {}
    PT operator + (const PT &p) const { return PT(x + p.x, y + p.y); }
    PT operator - (const PT &p) const { return PT(x - p.x, y - p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }
    bool operator < (const PT &p) const {
        if(cmp(x, p.x) != 0) return x < p.x;
        return cmp(y, p.y) < 0;
    }
    bool operator == (const PT &p) const { return !cmp(x, p.x) && !cmp(y, p
        .y); }
    bool operator != (const PT &p) const { return !(p == *this); }
};

// Debug function
ostream &operator << (ostream &os, const PT &p){
    os << "(" << p.x << "," << p.y << ")";
    return os;
}

// Function to calculate the dot product (u.v)
double dot (PT p, PT q) { return p.x * q.x + p.y*q.y; }

// Function to calculate the cross product (uXv) (2x2 determinant)
double cross (PT p, PT q) { return p.x * q.y - p.y*q.x; }

// Function to calculate the magnitude of the vector (|u|)
double norm (PT p) { return hypot(p.x, p.y); }
```

5.2 Convex Hull

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

#include "./Geometry.cpp"

// Monotone chain Algorithm to calculate Convex Hull
vector<PT> convexHull(vector<PT> p, bool needSort = 1){
    if(needSort) sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end()), p.end());
    int n = p.size(), k = 0;
    if(n <= 1) return p;
    vector<PT> answ(n + 2);   // Must be 2*n + 1 for collinear points

    // Lower hull
    for(int i = 0; i < n; i++){
        while(k >= 2 && cross(answ[k - 1] - answ[k - 2], p[i] - answ[k -
             2]) <= 0) k--; // If collinear points are allowed put only "<"
        answ[k++] = p[i];
    }

    // Upper hull
    for(int i = n - 2, t = k + 1; i >= 0; i--) {
        while(k >= t && cross(answ[k - 1] - answ[k - 2], p[i] - answ[k -
             2]) <= 0) k--; // If collinear points are allowed put only "<"
        answ[k++] = p[i];
    }

    answ.resize(k); // n+1 points where the first is equal to the last
    return answ;
}

/*

Time Complexity

convexHull  -> O(nlogn)

Links:

https://cp-algorithms.com/geometry/convex-hull.html#implementation_1
https://www.youtube.com/watch?v=JS-eBdqb1uM

*/
```

5.3 Polygon Area

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

// Calculation of polygon area using Shoelace Formula. (v -> (X,Y))
// Vertices need to be sorted in clockwise manner or anticlockwise from the
     first vertex to last.
double polygonArea(vector<pair<double,double>> v, int n){

    double area = 0.0;

    int j = n-1;
    for(int i = 0; i < n; i++){
        area += (v[j].first + v[i].first)*(v[j].second - v[i].second);
        j = i;  // j is previous vertex to i
    }

    return abs(area/2.0);   // Return absolute value
}

/*

Time Complexity

polygonArea     -> O(n)

Links:

https://www.geeksforgeeks.org/area-of-a-polygon-with-given-n-ordered-
    vertices/
```

```
*/
```

6 String Algorithms

6.1 Rabin Karp Hash

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

#define int long long

// Implementation of Rabin-Karp algorithm
class RabinKarp{
    const uint64_t MOD = (1LL << 61) - 1;
    const int base = 31;
    int n;
    vector<uint64_t> h;
    vector<uint64_t> p;

public:
    RabinKarp(string &s){ // Initializing
        this->n = s.size();
        p.assign(n, 0);
        h.assign(n, 0);

        p[0] = 1;
        h[0] = getInt(s[0]);
        for(int i = 1; i < n; i++){
            p[i] = modMul(p[i-1], base);
            h[i] = (modMul(h[i-1], base) + getInt(s[i])) % MOD;
        }
    }

    uint64_t getKey(int l, int r){   // [l, r]
        uint64_t answ = h[r];
        if(l > 0) answ = (answ + MOD - modMul(p[r - l + 1], h[l - 1])) %
            MOD;
        return answ;
    }

private:
    uint64_t getInt(char c){
        return c - 'a' + 1;
    }

    uint64_t modMul(uint64_t a, uint64_t b) {
        uint64_t l1 = (uint32_t)a, h1 = a >> 32, l2 = (uint32_t)b, h2 = b
            >> 32;
        uint64_t l = l1 * l2, m = l1 * h2 + l2 * h1, h = h1 * h2;
        uint64_t ret = (l & MOD) + (l >> 61) + (h << 3) + (m >> 29) + ((m
            << 35) >> 3) + 1;
        ret = (ret & MOD) + (ret >> 61);
        ret = (ret & MOD) + (ret >> 61);
        return ret - 1;
    }
};

/*

Time Complexity

RabinKarp   -> O(N)
getKey      -> O(1)

Links:

https://cp-algorithms.com/string/string-hashing.html#applications-of-
    hashing
https://cp-algorithms.com/string/rabin-karp.html
https://www.youtube.com/watch?v=qQ8vS2btsxI
https://codeforces.com/blog/entry/60445

*/
```

6.2 Z Algorithm

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

#define int long long

// Implementation of Z-function algorithm
class ZFunction{
private:
    int n;
    vector<int> z;
    int patternSize = -1;

public:
    ZFunction(string str, string pattern = ""){ // Initializing
        if(!pattern.empty()){
            str = pattern + "$" + str;
            this->patternSize = pattern.size();
        }
        this->n = str.size();
        z.assign(n, 0);

        int l = 0, r = 0;
        for(int i = 1; i < n; i++){ // Z-function
            if(i <= r){
                z[i] = min(r - i + 1, z[i - 1]);
            }
            while(i + z[i] < n && str[z[i]] == str[i + z[i]]) z[i]++;
            if(i + z[i] - 1 > r){
                l = i;
                r = i + z[i] - 1;
            }
        }
    }

    vector<int> findPattern(){
        vector<int> answ;
        for(int i = 0; i < n; i++){
            if(z[i] == patternSize) answ.push_back(i - patternSize - 1);
        }
        return answ;
    }
};

/*

Time Complexity

Z-function -> O(N)

Links:

https://cp-algorithms.com/string/z-function.html#efficient-algorithm-to-
    compute-the-z-function
https://www.youtube.com/watch?v=CpZh4eF8QBw

*/
```

7 Search Algorithms

7.1 Binary Search

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

// Recursive Binary Search function
int binarySearch(int arr[], int l, int r, int x){

    if (r >= l) {
        int mid = l + (r - l) / 2;

        // Found my answer
        if (arr[mid] == x)
            return mid;

        // Element is smaller, just need to look in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Element is greater, just need to look in left subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // Can't find element "x"
    return -1;
}

// Iterative Binary Search function
int binarySearch(int arr[], int l, int r, int x){

    while (l <= r) {
        int m = l + (r - l) / 2;

        // Found my answer
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // Can't find element "x"
    return -1;
}

// Finding the smallest solution
// Suppose that we wish to find the smallest value k that is a valid
     solution for a problem
// We know that check(x) is false when x < k and true when x >= k.
// The initial jump length "z" has to be large enough
int x = -1;
for(int b = z; b >= 1; b /= 2){
    while(!check(x+b)) x += b;
}
int k = x+1;

// Finding the maximum value
// BS can be used to find the maximun value for a function that is first
     increasing and then decreasing
// Not allowed that consecutive values of the function are equal.
int x = -1;
for(int b = z; b >= 1; b /=2){
    while(f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;

/*

Time Complexity

Binary Search                    -> O(logn)
Finding the smallest solution    -> O(x*logz) x is the time complexity of
     function check();
Finding the maximum value        -> O(x*logz) x is the time complexity of
     function f();

Links:

https://www.geeksforgeeks.org/binary-search/

*/
```

7.2 Ternary Search

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

// Algorithm for finding the maximum of f(x)  which is unimodal on an
     interval [l,r]
// Real numbers
double ternarySearch(double l, double r) {

    double eps = 1e-9;              // Set the error limit
    for(int i = 0; i < 200 && r-l > eps; i++){
        double m1 = (2*l + r)/3.0;
        double m2 = (l + 2*r)/3.0;

        if(f(m1) > f(m2))   //Evaluate Function "f" at m1 and m2
            l = m1;
        else
            r = m2;
    }

    return f(l);        // Return the maximum of f(x) in [l, r]
}

/*

Time Complexity

ternarySearch   -> O(log3(N))

Links:

https://cp-algorithms.com/num_methods/ternary_search.html

*/
```

8 Miscellaneous

8.1 Kadane

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

typedef long long ll;

vector<ll> v;
```

```
// Kadane s Algorithm (Works in array's that have only negative numbers)
ll kadane(int size){

    ll best = 0;
    ll currSum = 0;

    for(int i = 0; i < size; i++){

        // We have two option, ours current sum is actually starting at v[i
            ] or is the sum of previous subarray + v[i]
        currSum = max(v[i],currSum + v[i]);

        // Ours answer is the max between the previously answer or the
            current sum
        best = max(best,currSum);

    }

    return best;

}

// Kadane circular array (Method 1)
ll maxCircularSumMethod1(int size){

    // Ours answer don't have the corners (Just need to use Kadane)
    ll max_kadane = kadane(size);

    // if maximum sum using standard kadane' is less than 0
    if(max_kadane < 0)
      return max_kadane;

    // Ours answer can have the corners
    ll max_wrap = 0;

    for(int i = 0; i < size; i++) {
        max_wrap += v[i];   // Calculate array-sum
        v[i] = -v[i];       // invert the array (change sign)
    }

    // Max sum with corner elements will be:
    // array-sum - (-max subarray sum of inverted array)
    max_wrap = max_wrap + kadane(size);

    // The maximum circular sum will be maximum of two sums
    return max(max_wrap, max_kadane);
}

// Kadane circular array (Method 2)
ll maxCircularSumMethod2(int size){

    // Corner Case
    if (size == 1)
        return v[0];

    // Initialize sum variable which store total sum of the array.
    ll totalSum = 0;
    for (int i = 0; i < size; i++) {
        totalSum += v[i];
    }

    // Initialize every variable with first value of array.
    ll bestMax = v[0];
    ll currSumMax = v[0];

    ll bestMin = v[0];
    ll currSumMin = v[0];

    // Concept of Kadane's Algorithm
    for (int i = 1; i < size; i++) {
        // Kadane's Algorithm to find Maximum subarray sum.
        currSumMax = max(v[i], currSumMax + v[i]);
        bestMax = max(bestMax, currSumMax);

        // Kadane's Algorithm to find Minimum subarray sum.
        currSumMin = min(v[i], currSumMin + v[i]);
        bestMin = min(bestMin, currSumMin);
    }

    // All values are negative, just return bestMax
    if (bestMin == totalSum)
        return bestMax;

    // Else, we will calculate the maximum value
    return max(bestMax, totalSum - bestMin);
}

/*

Time Complexity

Kadane s Algorithm -> O(n)

Links:

https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/
https://www.geeksforgeeks.org/maximum-contiguous-circular-sum/

*/
```

8.2 Next Great Element

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

#define int long long

const int inf = 0x3f3f3f3f;
using ii = pair<int,int>;
using ll = long long;

const int N = 1e5; // Number of elements

vector<ll> v;

// NGE's answer from all elements in vector "v"
vector<ii> nxt(N,{inf,N});

// Find the Next Great Element for all array elements
// If an element does not exist it is defined as {inf, n}
void NGE(int n){

    // Push the first element {element, idx}
    stack<pair<ll,ll>> s;
    s.push({v[0],0});

    // Iterate for rest of the elements
    for (int i = 1; i < n; i++){

        if (s.empty()) {    // If the stack is empty just push the next
            element
            s.push({v[i],i});
            continue;
        }

        // if stack is not empty and the popped element is smaller than
            next (keep popping)
        while (s.empty() == false && s.top().first < v[i]){
            nxt[s.top().second] = {v[i],i};
            s.pop();
        }

        // Push "next" to stack so that we can find NGE for him
```

```
            s.push({v[i],i});
        }

        // The remaining elements in stack don't have NGE
        while (s.empty() == false) {
            nxt[s.top().second] = {inf,n};
            s.pop();
        }
    }

    // NGE with the fastest implementation using PGE's logic
    // If an element does not exist it is defined as {inf, n}
    void NGE(int n){

        // Push the first element {element, idx}
        stack<pair<ll,ll>> s;
        s.push({inf,n});    // The first element must be this

        // Traverse remaining elements (in reverse order of PGE)
        for(int i = n-1; i >= 0; i--){

            while (s.empty() == false && s.top().first < v[i])
                s.pop();

            if(s.empty())
                nxt[i] = {inf,n};
            else
                nxt[i] = {s.top().first, s.top().second};

            s.push({v[i],i});
        }
    }

    /*

    Time Complexity

    NGE -> O(n)

    Links:

    https://www.geeksforgeeks.org/next-greater-element/

    */
```

8.3 Previous Great Element

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

#define int long long

const int inf = 0x3f3f3f3f;
using ii = pair<int,int>;
using ll = long long;

const int N = 1e5; // Number of elements

vector<ll> v;

// PGE's answer from all elements in vector "v"
vector<ii> pre(N,{inf,-1});

// Find the previous great element for all array elements
// If an element does not exist it is defined as {inf,-1}
void PGE(int n){

    // Push the first element {element, idx}
    stack<pair<ll,ll>> s;
    s.push({v[0],0});

    // Traverse remaining elements
    for(int i = 1; i < n; i++){

        // Pop elements from stack while stack is not empty
        // and top of stack is smaller than arr[i]. We
        // always have elements in decreasing order in a
        // stack.
        while (s.empty() == false && s.top().first < v[i])
            s.pop();

        // If stack becomes empty, then no element is greater
        // on left side. Else top of stack is previous
        // greater.
        if(s.empty())
            pre[i] = {inf,-1};
        else
            pre[i] = {s.top().first, s.top().second};

        s.push({v[i],i});
    }
}

/*

Time Complexity

PGE -> O(n)

Links:

https://www.geeksforgeeks.org/previous-greater-element/

*/
```

8.4 Quick Select

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

#define int long long
using ll = long long;

vector<int> v;

int partition(int l, int r){
    int pivot = v[l + (r-l)/2];
    while(l < r){
        if(v[l] >= pivot){
            swap(v[l], v[r]);
            r--;
        }
        else
            l++;
    }

    if(v[l] < pivot) l++;

    return l;
}

// Algorithm to find the smallest "k"s elements using QuickSelect
// QuickSelect can be used to find the "k" element as well.
vector<int> smallest_k_elements(int k){
    int l = 0;
    int r = v.size()-1;
    int pivotIdx = v.size();

    while(pivotIdx != k){
        pivotIdx = partition(l,r);
        if(pivotIdx < k)
```

```
            l = pivotIdx;
        else
            r = pivotIdx - 1;
    }

    return vector<int>(v.begin(), v.begin() + k);
}

/*

Time Complexity

QuickSelect -> O(n) In the average case, in the worst case it runs on O(n
    ^2) but it is rare.

Links:

https://www.youtube.com/watch?v=ooLKYx1T1SE

*/
```

8.5 Spiral Traversal

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

typedef long long ll;

vector<ll> answ;

// Spiral Traversal (m = number of rows && n = number of columns)
// "v" is our matrix
void SpiralTraversal(int m, int n){

    int i, k = 0, l = 0;

    /*
        k - starting row index
        m - ending row index    (exclusive)
        l - starting column index
        n - ending column index (exclusive)
        i - iterator
    */

    while (k < m && l < n) {

        // Left to right
        for (i = l; i < n; ++i) {
            answ.push_back(v[k][i]);
        }
        k++;

        // From top to bottom
        for (i = k; i < m; ++i) {
            answ.push_back(v[i][n-1]);
        }
        n--;

        // Right to Left
        if (k < m) {
            for (i = n - 1; i >= l; --i) {
                answ.push_back(v[m-1][i]);
            }
            m--;
        }

        // From bottom to top
        if (l < n) {
            for (i = m - 1; i >= k; --i) {
                answ.push_back(v[i][l]);
            }
            l++;
        }
    }
}

/*

Time Complexity

SpiralTraversal -> O(m*n)

Links:

https://www.geeksforgeeks.org/print-a-given-matrix-in-spiral-form/

*/
```

9 Util

9.1 Structure for matrix

```
template<typename T = int>
struct Matrix{
    int r, c;
    vector<vector<T>> mat;

    Matrix(int r, int c): r(r), c(c){
        mat.assign(r, vector<T>(c, 0));
    }

    Matrix(vector<vector<T>> m, int r, int c): r(r), c(c){
        mat.assign(r, vector<T>(c, 0));
        for(int i = 0; i < r; i++)
            for(int j = 0; j < c; j++)
                this->mat[i][j] = m[i][j];
    }

    Matrix operator *(Matrix &other){
        Matrix answ(this->r, other.c);
        for(int i = 0; i < r; i++){
            for(int j = 0; j < other.c; j++){
                for(int k = 0; k < c; k++){ // MOD only if necessary
                    answ.mat[i][j] = (answ.mat[i][j] + mat[i][k] * other.
                        mat[k][j]%MOD)%MOD;
                }
            }
        }
        return answ;
    }

    vector<T> operator [](int r){
        return mat[r];
    }
};

/*

Using this struct with the following algorithms:

Time Complexity

fastModExp  -> O(d^3*logy)  d is the dimension of the square matrix

Links:

https://zobayer.blogspot.com/2010/11/matrix-exponentiation.html
https://www.geeksforgeeks.org/matrix-exponentiation/

*/
```