Planets and Kingdoms

A game has n planets, connected by m teleporters.

Two planets a and b belong to the same kingdom exactly when there is a route both
from a to b and from b to a.

Your task is to determine for each planet its kingdom.

Input

The first input line has two integers n and m: the number of planets and teleporters.
The planets are numbered 1,2,...,n.

After this, there are m lines describing the teleporters.
Each line has two integers a and b: you can travel from planet a to planet b through a teleporter.

Output

First print an integer k: the number of kingdoms.
After this, print for each planet a kingdom label between 1 and k.

You can print any valid solution.

Constraints

1 <= n <= 10^5
1 <= m <= 2.10^5
1 <= a,b <= n

Example

Input:
5 6
1 2
2 3
3 1
3 4
4 5
5 4

Output:
2
1 1 1 2 2

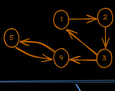
Problem Breakdown

Lets understand the problem first

We're given a directed graph with n planets (nodes) and m teleportation routes (edges).

A planet belongs to a kingdom, and every Strongly Connected Component (SCC) is a kingdom.

Our task is to find how many kingdoms (SCCs) there are, and assign each planet to one of them.



What is a Strongly Connected Component (SCC)?

A SCC is a maximal group of nodes where every node can reach every other node in the group.

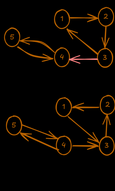
SCCs are the building blocks of DAGs (Directed Acyclic Graphs).

Why Kosaraju's Algorithm?

It's one of the fastest and simplest ways to find SCCs.

Runs in O(n + m) time — perfect for large graphs.

Kosaraju's Algorithm — 3 Steps



Step 1: DFS to get finishing order

Do a normal DFS from all unvisited nodes.

When you finish a node, push it to a stack.

Step 2: Reverse the graph

Flip all edges (i.e., if u → v, make it v → u).

Step 3: DFS on reversed graph (using stack order)

Pop nodes from the stack, and perform DFS on the reversed graph.

Each DFS gives you one SCC (i.e., one kingdom).

Algorithm

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

vector<vector<int>> graph, reversedGraph;
stack<int> finishOrder;
vector<bool> visited;

void dfsOriginal(int u) {
    visited[u] = true;
    for (int v : graph[u]) {
        if (!visited[v])
            dfsOriginal(v);
    }
    finishOrder.push(u);
}
```

Running DFS in random order
and filing stack according to their finish time

```cpp
void dfsReversed(int u, vector<int>& component) {
    visited[u] = true;
    component.push_back(u);
    for (int v : reversedGraph[u]) {
        if (!visited[v])
            dfsReversed(v, component);
    }
}
```

Running DFS in reversed graph
to get strongly connected components

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;

    graph.resize(n + 1);
    reversedGraph.resize(n + 1);

    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        reversedGraph[v].push_back(u);
    }
```

Reading input and creating
original graph and reversed graph

```cpp
    visited.assign(n + 1, false);
    for (int i = 1; i <= n; ++i) {
        if (!visited[i])
            dfsOriginal(i);
    }

    visited.assign(n + 1, false);
    vector<int> kingdom(n + 1, 0);
    int currentLabel = 1;

    while (!finishOrder.empty()) {
        int u = finishOrder.top();
        finishOrder.pop();

        if (!visited[u]) {
            vector<int> component;
            dfsReversed(u, component);
            for (int node : component) {
                kingdom[node] = currentLabel;
            }
            currentLabel++;
        }
    }
```

Running DFS in stack order

```cpp
    cout << currentLabel - 1 << '\n';
    for (int i = 1; i <= n; ++i) {
        cout << kingdom[i] << ' ';
    }
    cout << '\n';

    return 0;
}
```

Printing our answer

Time Complexity

O(n + m)

Space Complexity

O(n + m)