Building Teams

There are n pupils in Uolevi's class, and m friendships between them.

Your task is to divide the pupils into two teams in such a way that no two pupils in a team are friends.

You can freely choose the sizes of the teams.

Input

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

Then, there are m lines describing the friendships. Each line has two integers a and b: pupils a and b are friends.

Every friendship is between two different pupils. You can assume that there is at most one friendship between any two pupils.

Output

Print an example of how to build the teams. For each pupil, print "1" or "2" depending on to which team the pupil will be assigned.
You can print any valid team.

If there are no solutions, print "IMPOSSIBLE".

Constraints

1 <= n <= 10^5
1 <= m <= 2 · 10^5
1 <= a,b <= n

Example

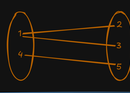
Input:
5 3
1 2
1 3
4 5
Output:
1 2 2 1 2

Problem Breakdown

Lets understand the problem first

You're given a graph of students (= nodes) and friendship relations (= edges).

Can you divide them into two teams such that no two friends are on the same team?



Recognize Graph Concepts

Each student = node

Each friendship = edge

2-team assignment = Bipartite graph

So, we have to use BFS to check bipartiteness.

What is a Bipartite Graph?

A graph is bipartite if you can:

Color all nodes using two colors (Team 1 and Team 2),

Such that no two connected nodes have the same color.

If it is possible to divide vertices into two mutually exclusive and exhaustive set such that all edges are across sets.

Algorithm (BFS Coloring)

1. Initialize all nodes as unvisited (team[i] = -1).

2. For every unvisited node:
   . Start a BFS.
   . Assign it to a team (say 1).
   . For each neighbor, assign it to the opposite team (2).
   . If a neighbor already has the same team → return impossible.

Acyclic Graph always bipartite

Cyclic Graph

Even Size
always bipartite

Odd size
(never bipartite)

Algorithm

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

#define endl '\n'

int n, m;
vector<vector<int>> graph;
vector<int> team; Used for coloring: 1 or 2

bool bfs(int start) {
    queue<int> q;
    q.push(start);
    team[start] = 1; // Team 1

    while (!q.empty()) {
        int node = q.front(); q.pop();

        for (int neighbor : graph[node]) {   Assign opposite team to neighbors
            if (team[neighbor] == -1) {
                team[neighbor] = 3 - team[node]; // Switch between 1 and 2   Neat trick to toggle between 1 and 2
                q.push(neighbor);
            } else if (team[neighbor] == team[node]) {
                return false; // Conflict → same team for adjacent nodes   If adjacent nodes have same team, not bipartite
            }
        }
    }

    return true;
}

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

    cin >> n >> m;

    graph.resize(n + 1);
    team.assign(n + 1, -1);

    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;

        graph[u].push_back(v);
        graph[v].push_back(u);
    }   Adjacency list for undirected graph

    for (int i = 1; i <= n; ++i) {
        if (team[i] == -1) {
            if (!bfs(i)) {
                cout << "IMPOSSIBLE" << endl;
                return 0;
            }
        }
    }

    for (int i = 1; i <= n; ++i) {
        cout << team[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Time Complexity

O(N + M)

we visit every cell at most once.

Space Complexity

O(N + M)

for queues and auxiliary arrays

Generalization

Bipartiteness is useful in:

- Scheduling problems
- Matching problems
- Alternating path questions

This idea generalizes to:

- 2-coloring
- Cycle detection
- Graph coloring

```
bool dfs(int node, int currentTeam) {
    team[node] = currentTeam;

    for (int neighbor : graph[node]) {
        if (team[neighbor] == -1) {
            // Assign opposite team to neighbor
            if (!dfs(neighbor, 3 - currentTeam)) return false;
        } else if (team[neighbor] == currentTeam) {
            // Conflict: neighbor is on same team
            return false;
        }
    }

    return true;
}
```