Round Trip

Byteland has n cities and m roads between them.

Your task is to design a round trip that begins in a city, goes through two or more other cities, and finally returns to the starting city.

Every intermediate city on the route has to be distinct.

Input

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

Then, there are m lines describing the roads. Each line has two integers a and b: there is a road between those cities.

Every road is between two different cities, and there is at most one road between any two cities.

Output

First print an integer k: the number of cities on the route. Then print k cities in the order they will be visited. You can print any valid solution.
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 6
1 3
1 2
5 3
1 5
2 4
4 5
```
Output:
```
4
3 5 1 3
```

Problem Breakdown

Lets understand the problem first

Given an undirected graph, find and print any cycle (round trip).

If no cycle exists, print "IMPOSSIBLE".

This is a classic graph cycle detection problem.

Recognize Graph Concepts

Each city = node

Each road = edge

Cycle(round trip) = A path that starts and ends at the same node

What's the core idea?

Use DFS to traverse the graph.

Mark visited nodes.

If you find an already visited node that is not your parent, then you found a cycle.

Algorithm (DFS)

Steps:

1. Use DFS from every unvisited node.
2. Keep a visited[] array.
3. Track the parent of each node (parent[]).
4. If you reach an already visited node (not the parent), you found a cycle.
5. Use parent[] to backtrack the cycle path.

Algorithm

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

#define endl '\n'

int n, m;
vector<vector<int>> graph;
vector<int> parent; used to reconstruct the cycle
vector<bool> visited;
int cycle_start = -1, cycle_end; to mark where the cycle begins and ends

bool dfs(int node, int par) {
    visited[node] = true;
    for (int neighbor : graph[node]) {

        if (neighbor == par) continue;  // don't go back via the same edge

        if (visited[neighbor]) {
            cycle_start = neighbor;
            cycle_end = node;
            return true;
        } else {
            parent[neighbor] = node;
            if (dfs(neighbor, node)) return true;
        }
    }
    return false;
}

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

    cin >> n >> m;
    graph.resize(n + 1);
    parent.assign(n + 1, -1);
    visited.assign(n + 1, false);

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

    for (int i = 1; i <= n; ++i) {
        if (!visited[i] && dfs(i, -1)) break;
    }

    if (cycle_start == -1) {
        cout << "IMPOSSIBLE" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        int curr = cycle_end;
        while (curr != cycle_start) {
            cycle.push_back(curr);
            curr = parent[curr];
        }
        cycle.push_back(cycle_start);  // close the cycle

        reverse(cycle.begin(), cycle.end());  // to print in correct order

        cout << cycle.size() << endl;
        for (int node : cycle) {
            cout << node << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Performs DFS and detects a cycle

Adjacency list for undirected graph

Runs DFS on disconnected components too

Constructing cycle

Time Complexity

O(N + M)

we visit every cell at most once.

Space Complexity

O(N + M)

Common Questions

Why check neighbor != parent?	To ignore the edge that leads back to parent
Why not use BFS?	You can, but DFS is easier for path backtracking
What if the graph is disconnected?	We check each component
Why reverse the cycle at the end?	Because we trace it backwards using parent[]
Can there be more than one cycle?	Yes, but the problem only asks for any one

Generalization

You can use this logic to solve:

- Cycle detection in other types of graphs
- Tree validation
- Connected components with cycles
- DFS with back edges