# Round Trip II

Byteland has n cities and m flight connections.

Your task is to design a round trip that begins in a city, goes through one 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 flights.
The cities are numbered 1,2,...,n.

Then, there are m lines describing the flights.
Each line has two integers a and b: there is a flight connection from city a to city b. All connections are one-way flights from a city to another city.

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.

If there are no solutions, print "IMPOSSIBLE".

Constraints

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

Example

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

Output:
4
2 1 3 2

Problem Breakdown

Lets understand the problem first

Given a directed graph, find and print any cycle (round trip).

If no cycle exists, print "IMPOSSIBLE".

This is a classic graph cycle detection problem in directed graph.


How is different from cycle detection in undirected graph

Previously, we were just checking if the neighbour of current node is

-> visited
-> and not parent

we got our cycle


The Core Idea:

To detect a cycle during DFS, we need to know the path (chain) we've taken to reach the current node.

Why?

Because if we find a neighbor that's already in our current DFS path, that means we're revisiting a node on the same path → a cycle is found!

Option 1: Use a Set to Store the DFS Path

A naive way is to maintain a set that keeps track of all nodes in the current DFS path (chain).

We add a node to the set when we enter it.

Remove it when we backtrack.

Then, for every neighbor v of u, if v is already in the set, we've found a cycle.

✅ This works.
❌ But it's inefficient (set operations are O(log n)) and doesn't help us reconstruct the actual cycle path easily.

Better Solution: Use a Recursion Stack Tracker

Instead of a set, we use a boolean array recStack[] to simulate the current DFS call stack:

recStack[u] = true when we enter node u.

recStack[u] = false when we leave it.

Now, if we ever encounter a neighbor v such that recStack[v] == true, we've found a cycle!

Role of Other Arrays:

visited[]: Tracks if a node has ever been visited before (standard in DFS).

parent[]: Helps reconstruct the path once we find a cycle. For every node, we store the node it came from.

cycleStart and cycleEnd: We save these two when we detect a cycle so we can backtrack and print the cycle path.

Summary of the Approach:

1. Start DFS from every unvisited node.
2. While doing DFS:
   -> Mark the current node in both visited[] and recStack[].
   -> For each neighbor:
      -> If it's unvisited → continue DFS.
      -> If it's already in recStack[] → cycle found.
3. If cycle is found:
   -> Use parent[] to backtrack and reconstruct the cycle path.
4. If no cycle is found in the entire graph → print "IMPOSSIBLE".

# Algorithm

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

int n, m;
vector<vector<int>> graph;
vector<bool> visited;
vector<bool> recStack;
vector<int> parent;
int cycleStart = -1;
int cycleEnd = -1;

bool dfs(int u) {
    visited[u] = true;
    recStack[u] = true;

    for (int v : graph[u]) {
        if (!visited[v]) {
            parent[v] = u;
            if (dfs(v))
                return true;
        } else if (recStack[v]) {
            cycleEnd = u;
            cycleStart = v;
            return true;
        }
    }

    recStack[u] = false;
    return false;
}

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

    cin >> n >> m;

    graph.resize(n + 1);
    visited.resize(n + 1, false);
    recStack.resize(n + 1, false);
    parent.resize(n + 1, -1);

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

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

    if (cycleStart == -1) {
        cout << "IMPOSSIBLE" << endl;
        return 0;
    }

    vector<int> cycle;
    cycle.push_back(cycleStart);
    for (int v = cycleEnd; v != cycleStart; v = parent[v]) {
        cycle.push_back(v);
    }
    cycle.push_back(cycleStart);
    reverse(cycle.begin(), cycle.end());

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

    return 0;
}
```

DFS for Cycle Detection

Mark the current node in both visited[] and recStack[].

For each neighbor:

If it's unvisited → continue DFS.

If it's already in recStack[] → cycle found.

Graph Representation: The graph is represented using an adjacency list where each node points to its neighbors.

Cycle Reconstruction: Once a cycle is detected, the path is reconstructed by backtracking from the end of the cycle to the start using the parent array.

Time Complexity

O(N + M)
Each node and edge is visited once during DFS.

Space Complexity

O(N)
For visited, recStack, and parent arrays.

## Common Questions

| | |
|---|---|
| Why not just use visited[] for detecting cycles? | Because you may revisit nodes from another path. You need to check if the node is in the current DFS path. |
| What is a back edge? | An edge pointing to an ancestor in the DFS path — signals a cycle. |
| Why do we need to check from all nodes? | Because the graph may have disconnected components. |
| Can we use BFS here? | No, BFS doesn't track recursion stack; not suitable for this type of cycle detection. |