Longest Flight Route

Uolevi has won a contest, and the prize is a free flight trip that can consist of one or more flights through cities. Of course, Uolevi wants to choose a trip that has as many cities as possible.

Uolevi wants to fly from Syrjälä to Lehmälä so that he visits the maximum number of cities. You are given the list of possible flights, and you know that there are no directed cycles in the flight network.

Input

The first input line has two integers n and m: the number of cities and flights.
The cities are numbered 1,2,...,n. City 1 is Syrjälä, and city n is Lehmälä.

After this, there are m lines describing the flights. Each line has two integers a and b: there is a flight from city a to city b. Each flight is a one-way flight.

Output

First print the maximum number of cities on the route. After this, print the cities in the order they will be visited.
You can print any valid solution.

If there are no solutions, print "IMPOSSIBLE".

Constraints

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

Example

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

Output:
4
1 3 4 5

Problem Breakdown

Lets understand the problem first

-> have to go from 1 to n via longest route

Let's say we are standing at some node v.
what do we really want to know?

🎯 what is the longest path from node 1 to node v?

This means for each node v, we want to find the maximum number of edges (or steps) it takes to reach there from node 1.

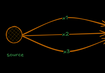
This immediately sounds like Dynamic Programming.
why?

Because we are solving a subproblem for each node:

what's the longest path to reach this node?

So our dp state becomes:

dp[v] = length of longest path from node 1 to node v



Why Not Use DFS or BFS Directly?

You might think:

Let's just run DFS or BFS from node 1 and relax every neighbor.

But here's the problem:

In a DAG, the graph may have multiple dependencies.

If you don't process the nodes in the correct order, you might try to compute dp[v] before dp[u] is finalized, even if u -> v is an edge.

❌ That gives wrong results.

We Need a Valid Processing Order

So what do we need?

We want an order where for every edge u -> v,
u is always processed before v.

This is exactly what Topological Sorting gives us.

✅ Topological Sort ensures that:

All dependencies are handled before we move to the next node.

Perfect for applying DP over DAGs.

Algorithm Overview

1. Topological Sort the graph.

2. Initialize dp[1] = 1, because path to itself is length 1.

3. For every node u in topological order:

-> For each neighbor v of u:

-> If going from u to v gives a longer path:

-> Update dp[v] = dp[u] + 1

-> Track parent[v] = u to reconstruct the path.

4. After processing:

-> If dp[n] == 0, that means node n was never reached → IMPOSSIBLE.

-> Else, we reconstruct the path using the parent array.

Algorithm

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

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

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

    vector<vector<int>> graph(n + 1);

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

```
    // Step 1: Topological Sort using Kahn's Algorithm
    vector<int> indeg(n + 1, 0);
    for (int u = 1; u <= n; ++u) {
        for (int v : graph[u]) {
            indeg[v]++;
        }
    }

    queue<int> q;
    vector<int> order;

    for (int i = 1; i <= n; ++i) {
        if (indeg[i] == 0) {
            q.push(i);
        }
    }

    while (!q.empty()) {
        int u = q.front(); q.pop();
        order.push_back(u);
        for (int v : graph[u]) {
            indeg[v]--;
            if (indeg[v] == 0) {
                q.push(v);
            }
        }
    }
```

Kahn's Algorithm

```
    // Step 2: Longest path DP on topological order
    vector<int> dp(n + 1, INT_MIN); // distance
    vector<int> parent(n + 1, -1);
    dp[1] = 0;

    for (int u : order) {
        if (dp[u] == INT_MIN) continue;
        for (int v : graph[u]) {
            if (dp[v] < dp[u] + 1) {
                dp[v] = dp[u] + 1;
                parent[v] = u;
            }
        }
    }
```

Longest Path DP

Initialize dp[1] = 0 (starting node)

Process nodes in topological order

For each neighbor, relax the distance:

```
    // Step 3: Check if path exists
    if (dp[n] == INT_MIN) {
        cout << "IMPOSSIBLE\n";
        return 0;
    }
```

```
    // Step 4: Reconstruct the path
    vector<int> path;
    int curr = n;
    while (curr != -1) {
        path.push_back(curr);
        curr = parent[curr];
    }

    reverse(path.begin(), path.end());

    cout << path.size() << '\n';
    for (int node : path) {
        cout << node << ' ';
    }
    cout << '\n';

    return 0;
}
```

Path Reconstruction

Check if dp[n] was reached

Backtrack from n using parent array

Reverse to get correct order

Time Complexity

Topological Sort: O(n + m)

DP Relaxation: O(m)

Path Reconstruction: O(n)

Overall: O(n + m)

Space Complexity

O(n + m) for graph + O(n) for dp, parent, etc.