Course Name:	**Competitive Programming Laboratory (216U01L401)**	**Semester:**	**IV**
Date of Performance:	___ / ___ / _____	**DIV/ Batch No:**	**A 3**
Student Name:	**Anushrut Pandit**	**Roll No:**	**16010123056**

Experiment No: 1

Title: To implement and apply BFS and DFS to solve graph-based competitive programming problems.

Aim and Objective of the Experiment:

1. Understand the concepts of BFS and DFS
2. Apply the BFS/DFS concepts to solve the graph problem
3. Implement the solution to given problem statement
4. Analyze the result for efficiency

COs to be achieved:

CO2: Analyze and optimize algorithms using amortized analysis and bit manipulation, equipping them to tackle complex computational problems.

Books/ Journals/ Websites referred:

1. Students should write

Theory:

The problem requires us to find a round trip in an undirected graph. A round trip is essentially a cycle where the starting and ending cities are the same. We need to check if such a cycle exists and, if so, print the sequence of cities forming the cycle.

Approach:
1. Graph Representation: The cities and roads are represented using an adjacency list.
2. Cycle Detection Using DFS: We traverse the graph using Depth-First Search (DFS) while keeping track of visited nodes and their parents to avoid backtracking.
3. Finding the Cycle:
- If we encounter a previously visited node that is not the immediate parent, we have found a cycle.
- We then reconstruct the cycle path using the parent array.
- If a cycle is found, we print the cities forming the round trip.
- If no cycle exists, we output "IMPOSSIBLE".

Complexity:
The algorithm runs in O(n + m) time complexity since DFS visits each node and edge at most once.

Problem statement

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. 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 5 6 1 3 1 2 5 3 1 5 2 4 4 5 Output: 4 3 5 1 3

Code :

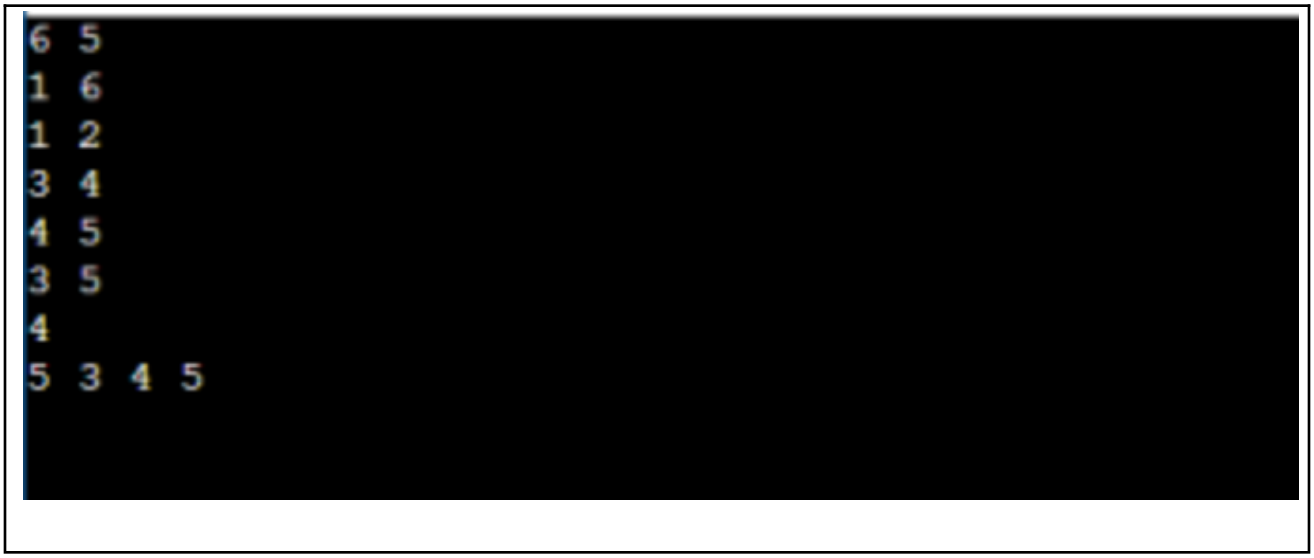
```
#include <bits/stdc++.h>
using namespace std;

const int MAX_N = 1e5 + 5;
vector<int> adj[MAX_N];
vector<int> parent(MAX_N, -1);
vector<bool> visited(MAX_N, false);
int start = -1, end = -1;

bool dfs(int node, int par) {
        visited[node] = true;
        for (int neighbor : adj[node]) {
        if (neighbor == par) continue;
        if (visited[neighbor]) {
        start = neighbor;
        end = node;
        return true;
        }
        parent[neighbor] = node;
        if (dfs(neighbor, node)) return true;
        }
```

```
        return false;
}

int main() {
        ios::sync_with_stdio(false);
        cin.tie(0);

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

        for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
        }

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

        if (start == -1) {
        cout << "IMPOSSIBLE" << endl;
        } else {
        vector<int> cycle;
        cycle.push_back(start);
        for (int v = end; v != start; v = parent[v]) {
        cycle.push_back(v);
        }
        cycle.push_back(start);
        reverse(cycle.begin(), cycle.end());

        cout << cycle.size() << "\n";
        for (int city : cycle) {
        cout << city << " ";
        }
        cout << "\n";
        }
```

```
        return 0;
}
```

Output:

```
10 20
9 8
9 5
6 4
5 10
7 5
7 8
3 4
6 5
2 1
10 4
6 1
9 7
7 3
4 5
2 9
5 3
2 3
8 5
6 7
3 8
5
5 9 8 7 5
```


6 5
1 6
1 2
3 4
4 5
3 5
4
5 3 4 5