Course Name:	**Competitive Programming Laboratory (216U01L401)**	**Semester:**	**IV**
Date of Performance:	**27 / 03 / 2025**	**DIV/ Batch No:**	**A1**
Student Name:	**Aaryan Sharma**	**Roll No:**	**16010123012**

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. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
2. cses.fi

Theory:

BFS (Breadth-First Search): BFS explores nodes level by level, visiting all neighbors of a node before moving deeper. It uses a queue to keep track of the next node to visit. It is useful for shortest path problems in an unweighted graph. Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges.
DFS (Depth-First Search): DFS explores as far as possible along one branch before backtracking. It uses a stack (or recursion) to keep track of the visited nodes. DFS helps in cycle detection, topological sorting, and connected component identification. Time Complexity: O(V + E).

Problem statement

1.Implementation of BFS with queue displayed
2. Implementation of DFS with stack displayed

3. Implementation of 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".

Example

Input:

5 6

1 3

1 2

5 3

1 5

2 4

4 5

Output:

4

3 5 1 3

4. Implementation of Round Trip 2: 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 mmm 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".

Example

Input:

4 5

1 3

2 1

2 4

3 2

3 4

Output:

4

2 1 3 2

Code:

1.

```
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;

bool DFS(vector<int> graph[], int s, vector<bool> &visited, vector<int> &parent,
vector<int> &cycle)
{
  visited[s] = true;
  for (int x : graph[s])
  {
    if (!visited[x])
    {
      parent[x] = s;
      if (DFS(graph, x, visited, parent, cycle))
      {
        return true;
      }
    }
    else if (x != parent[s])
    {
      cycle.push_back(s);
      int current = s;
      while (current != x)
      {
        current = parent[current];
        cycle.push_back(current);
      }
      reverse(cycle.begin(), cycle.end());
      cycle.push_back(x);
      return true;
    }
  }
  return false;
}

int main()
{
  int n, m;
  cin >> n >> m;
  vector<int> graph[n + 1];
  for (int i = 0; i < m; i++)
```

```
  {
    int a, b;
    cin >> a >> b;
    graph[a].push_back(b);
    graph[b].push_back(a);
  }

  vector<bool> visited(n + 1, false);
  vector<int> parent(n + 1, -1);
  vector<int> cycle;
  for (int i = 1; i <= n; i++)
  {
    if (!visited[i])
    {
      if (DFS(graph, i, visited, parent, cycle))
      {
        cout << cycle.size() << endl;
        for (int city : cycle)
        {
          cout << city << " ";
        }
        cout << endl;
        return 0;
      }
    }
  }
  cout << "IMPOSSIBLE" << endl;
}
```

2.

```
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;

bool found = false;
vector<int> cycle;

bool dfs(int u, vector<vector<int>> &graph, vector<bool> &visited, vector<bool>
&in_stack, vector<int> &path)
{
  visited[u] = true;
  in_stack[u] = true;
  path.push_back(u);
```

```
  for (int v : graph[u])
  {
    if (!visited[v])
    {
      if (dfs(v, graph, visited, in_stack, path))
      {
        return true;
      }
    }
    else if (in_stack[v])
    {
      auto it = find(path.begin(), path.end(), v);
      if (it != path.end())
      {
        cycle = vector<int>(it, path.end());
        cycle.push_back(v);
        found = true;
        return true;
      }
    }
  }
  in_stack[u] = false;
  path.pop_back();
  return false;
}

int main()
{
  int n, m;
  cin >> n >> m;
  vector<vector<int>> graph(n + 1);
  for (int i = 0; i < m; ++i)
  {
    int a, b;
    cin >> a >> b;
    graph[a].push_back(b);
  }

  vector<bool> visited(n + 1, false);
  vector<bool> in_stack(n + 1, false);
  vector<int> path;
  for (int u = 1; u <= n && !found; ++u)
  {
    if (!visited[u])
```

```
    {
      if (dfs(u, graph, visited, in_stack, path))
      {
        break;
      }
    }
  }

  if (found)
  {
    cout << cycle.size() << endl;
    for (size_t i = 0; i < cycle.size(); ++i)
    {
      if (i > 0)
        cout << " ";
      cout << cycle[i];
    }
    cout << endl;
  }
  else
  {
    cout << "IMPOSSIBLE" << endl;
  }
}
```

Output:

1.

Sample Input

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

Your Output

```
4
1 3 5 1
```

2.

Sample Input

```
4 5
1 3
2 1
2 4
3 2
3 4
```

Your Output

```
4
1 3 2 1
```

Conclusion:

I have implemented BFS and DFS to solve graph-based competitive programming problems. Through this experiment, I understood the differences between both traversal techniques and their applications. Additionally, I explored cycle detection using DFS for both directed and undirected graphs.