Mail Delivery

Your task is to deliver mail to the inhabitants of a city.

For this reason, you want to find a route whose starting and ending point are the post office, and that goes through every street exactly once.

Input

The first input line has two integers n and m: the number of crossings and streets.
The crossings are numbered 1,2,...,n, and the post office is located at crossing 1.

After that, there are m lines describing the streets.
Each line has two integers a and b: there is a street between crossings a and b. All streets are two-way streets.

Every street is between two different crossings, and there is at most one street between two crossings.

Output

Print all the crossings on the route in the order you will visit them. 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:

```
6 8
1 2
1 3
2 3
2 4
2 6
3 5
3 6
4 5
```

Output:

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

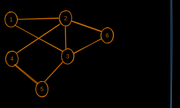
Problem Breakdown

Lets understand the problem first

You're given n crossings and m roads.

Your goal is to find a path such that we start from post office travel to all the crossings using streets and come back to the post office at last.

Each street can only be used once



1 -> 2 -> 6 -> 3 -> 2 -> 4 -> 5 -> 3 -> 1

Eulerian Path and Circuit

Think of a graph as a network of roads (edges) connecting cities (vertices).

Eulerian Path:
A walk that travels through every road exactly once.
You can pass through cities more than once, but you can't reuse any road.
Start and end points can be different.

Eulerian Circuit (or Cycle):
A special Eulerian Path that starts and ends at the same city.

Rules for Undirected Graphs

Here, roads have no direction — you can go either way.

Eulerian Circuit exists if
The graph is connected (ignoring isolated vertices), and
Every vertex has an even number of roads connected to it (even degree).

Eulerian Path exists if

The graph is connected (again ignoring isolated vertices), and
Exactly 0 or 2 vertices have an odd number of roads.

0 → it's actually an Eulerian Circuit.
2 → the path starts at one odd-degree vertex and ends at the other.

Rules for Directed Graphs

Here, roads have arrows (one-way).

Eulerian Circuit exists if
The graph is strongly connected (you can reach any vertex from any other following directions), and

For every vertex: in-degree = out-degree (roads coming in = roads going out).

Eulerian Path exists if
The graph is connected in a "weak" sense (ignoring directions, it's connected), and

Exactly one vertex has (out-degree − in-degree) = 1 (start),

Exactly one vertex has (in-degree − out-degree) = 1 (end),

All other vertices have in-degree = out-degree.

How to solve our problem we have to

Find a Eulerian cycle we run a modified DFS for it.

The DFS goes through only unvisited edges and the same edge can be processed multiple times through the DFS, so we remove it from the graph at the first visit.

Also we will give each edge an index so to know if it is visited or not
and at last while backtracking we will add nodes to our path


```
void dfs(int node) {
    while(!graph[node].empty()) {
        auto edge = graph[node].back();
        graph[node].pop_back();
        if(seen[edge.second]) continue;
        seen[edge.second] = true;
        dfs(edge.first);
    }
    path.push_back(node);
}
```

Algorithm

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

vector<vector<pair<int, int>>> graph;
vector<int> deg;
vector<bool> seen;
vector<int> path;

void dfs(int node) {
    while(!graph[node].empty()) {
        auto edge = graph[node].back();
        graph[node].pop_back();
        if(seen[edge.second]) continue;
        seen[edge.second] = true;
        dfs(edge.first);
    }
    path.push_back(node);
}

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

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

    graph.resize(n+1);
    deg.resize(n+1, 0);

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

    for(int i=1; i<=n; ++i) {
        if(deg[i] % 2 != 0) {
            cout << "IMPOSSIBLE" << endl;
            return 0;
        }
    }

    seen.resize(m, false);
    path.clear();

    dfs(1);

    if(path.size() != m + 1) {
        cout << "IMPOSSIBLE" << endl;
    } else {
        for(int i=path.size() - 1; i >= 0; --i) {
            cout << path[i] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Modified DFS to track the path

Constructing graph

Checking if degree is odd if yes we return

Checking if graph is connected if yes printing the result

Time Complexity

O(V + M)

Space Complexity

O(V + M)