Course Schedule

You have to complete n courses. There are m requirements of the form "course a has to be completed before course b". Your task is to find an order in which you can complete the courses.

Input

The first input line has two integers n and m: the number of courses and requirements.
The courses are numbered 1,2,...,n.

After this, there are m lines describing the requirements.
Each line has two integers a and b: course a has to be completed before course b.

Output

Print an order in which you can complete the courses.
You can print any valid order that includes all the courses.

If there are no solutions, print "IMPOSSIBLE".

Constraints

1 <= n <= 10^5
1 <= m <= $2\cdot10^5$
1 <= a,b <= n

Example

Input:
5 3
1 2
3 1
4 5

Output:
3 4 1 5 2

Problem Breakdown

Lets understand the problem first

Imagine you're planning your semester courses. Some courses require prerequisites. For example:

You can't study Algorithms unless you've completed Data Structures.

You can't do Machine Learning unless you've done Linear Algebra and Probability.

Now you want to figure out:

Can you finish all courses?

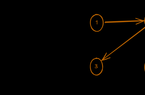
If yes, in what order?

💡 This is exactly what topological sort answers!

Topological Sort

Topological sorting is a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u → v), vertex u comes before vertex v in the ordering.

It is used in scheduling tasks, dependency resolution, and more.



Using DFS:-

Push nodes to a stack only after all their descendants have been visited.

The stack (when reversed) gives the topological order.

Algorithm Steps

1. Initialize visited and recursionStack arrays.
2. For each unvisited node, call DFS.
3. In DFS:
   -> Mark the node as visited.
   -> Recursively visit all unvisited neighbors.
   -> After processing all neighbors, push the node into a stack.
4. Reverse the stack to get the topological order.

Code Example (DFS-Based Topological Sort)

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

vector<vector<int>> graph;
vector<bool> visited;
stack<int> order;

void dfs(int u) {
    visited[u] = true;
    for (int v : graph[u]) {
        if (!visited[v]) {
            dfs(v);
        }
    }
    order.push(u);
}

vector<int> topologicalSort(int n) {
    visited.assign(n + 1, false);
    for (int u = 1; u <= n; ++u) {
        if (!visited[u]) {
            dfs(u);
        }
    }
    vector<int> result;
    while (!order.empty()) {
        result.push_back(order.top());
        order.pop();
    }
    return result;
}
```

Push nodes in stack in post-order to maintain dependencies

Reverse stack at last to get correct order

Complexity

Time: O(V + E) (each node and edge is processed once)

Space: O(V) (for recursion stack and visited array)

If we have to detect cycle using dfs approach we have to do it similar to the cycle detection in directed graphs like we did in the problem Round Trip 2

Why we can't push in pre-order?

Postorder DFS ensures that all dependencies (v) are processed before the dependent node (u).

Preorder DFS processes u before its dependencies, leading to incorrect ordering.

Using Kahn's Algorithm (BFS-Based)

1. Compute in-degree (number of incoming edges) for each node.
2. Enqueue nodes with in-degree 0 (no dependencies).
3. Process nodes:
   -> Remove a node from the queue and add it to the topological order.
   -> Decrease the in-degree of its neighbors.
   -> If a neighbor's in-degree becomes 0, enqueue it.
4. If the topological order has all nodes, it's valid; otherwise, the graph has a cycle.

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);
        }

        vector<int> indeg(n+1, 0);
        for(int u = 1; u <= n; ++u) {
                for(int v : graph[u]) {
                        indeg[v]++;
                }
        }

        queue<int> q;

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

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

        if(res.size() == n) {
                for(int node : res) {
                        cout << node << " ";
                }
                cout << endl;
                return 0;
        }

        cout << "IMPOSSIBLE" << endl;
}
```

Reading input and creating adjacency list(graph)

Count of prerequisites for each node

Count incoming edges

Start with nodes with no prerequisites (in-degree == 0)

While the queue is not empty:

Dequeue a node and add it to the topological order.

Reduce the in-degree of its neighbors.

If a neighbor's in-degree becomes 0, enqueue it.

Time Complexity

O(V + E) (each node and edge is processed once)

Space Complexity

O(V) (for queue and in-degree storage)