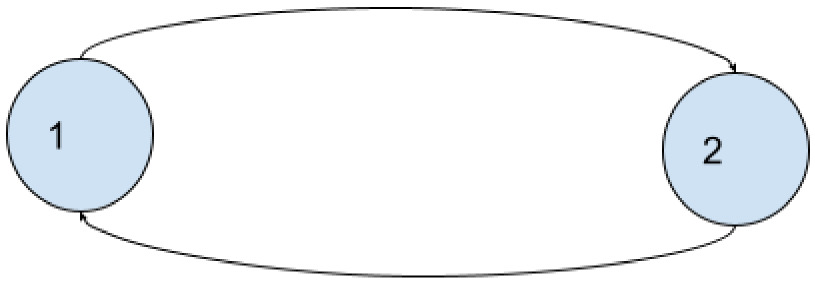
MEDIUM

Course Schedule | Course Schedule-||

Intuition

A deadlock is a specific type of concurrency-related problem that can occur in a multi-threaded or multi-process environment, where two or more threads or processes become blocked, each waiting for a resource that the other(s) holds, effectively preventing any of them from making progress. In other words, it's a situation in which multiple entities are unable to proceed because each is waiting for another to release a resource.

If there is a course that needs to be done before another and the course requires prior completion of an incoming task we enter a deadlock this means that we can not complete the course.



1 needs to be completed before 2

2 needs to be completed before 1

This is a deadlock and we can not complete task

We will be using the topological sort algorithm to detect if there is a cycle in the given graph and this will get us the result.

Eg.
n = 4

p = 3

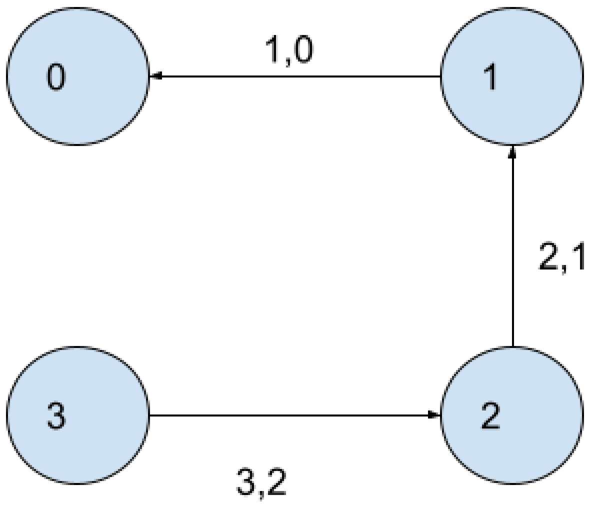
[[1,0], [2, 1], [3, 2]]

0 - 1

1 - 1

2 - 1

3 - 0



Adjacency list :

0 - [ ]

1 - [1]

2 - [1]

3 - [2]

Indegree = [1 | 1 | 3 | 0] = [1 | 1 | 0 | 0]

Q [3]

Q [2]

Q [1]

Q [0]

Result = [3 2 1 0]

Returns [0 1 2 3]

We will return the topological order if the completion is possible ie. reverse of traversal and check if the traversal size is equal to that of the number of nodes will tell us if the completion is possible or not

Approach

```
isPossible( )
    -   Create an adjacency list
    -   Return negation of cycle detection function as
            -   Return !detect(n,adj)
detect( )
    -   Create a vector to store the indegree of elements
    -   Initialize the indegree vector
    -   Declare :
            -   An empty queue
            -   A result vector
    -   Initialize queue with the elements having 0 indegree
    -   Traverse until the queue becomes empty :
            -   Extract and pop the first element of the queue
            -   Append the element
            -   Traverse for the adjacent elements of the node :
                    -   Reduce the indegree of the adjacent node
                    -   Check if the indegree of adjacent node becomes 0 then append it to
                        queue
    -   return result vector
```

Function Code

```
vector<int> detect(int n, vector<vector<int>>& adj) {
    // creating a vector for indegree
    vector<int> indegree(n, 0);
    // initializing indegrees
    for (int i = 0; i < n; i++) {
        for (int it : adj[i]) {
            indegree[it]++;
        }
    }

    // creating an empty queue
    queue<int> q;
    // adding nodes with indegree 0 to the queue
    for (int i = 0; i < n; i++) {
        if (indegree[i] == 0) {
            q.push(i);
        }
    }

    // creating a result vector
    vector<int> result;
    // traversing until the queue becomes empty
    while (!q.empty()) {
        // extracting first node and popping it
        int node = q.front();
        q.pop();

        // increasing the counter variable
        result.push_back(node);

        // traversing for the adjacent nodes
        for (int i : adj[node]) {
            // reducing indegree of node
            indegree[i] -= 1;
            // checking if indegree becomes 0
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
    }

    return result;
}

    vector<int> findOrder(int n,int p,vector<vector<int>>& prerequisites) {

        // creating an adjacency list
        vector<vector<int>> adj(n);
        for(int i=0;i<prerequisites.size();i++)
        {
            adj[prerequisites[i][0]].push_back(prerequisites[i][1]);
        }
        vector<int> result = detect(n,adj);
        if(result.size()==n)
        {
            reverse(result.begin(),result.end());
            return result;
        }
        return {};
    }
```

Time Complexity

$O(V+E)$