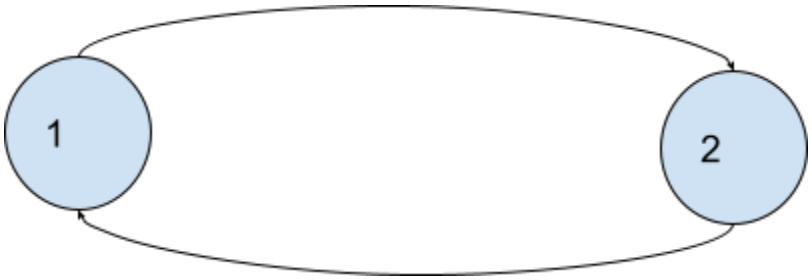
MEDIUM

Course Schedule | Prerequisite Task - I

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

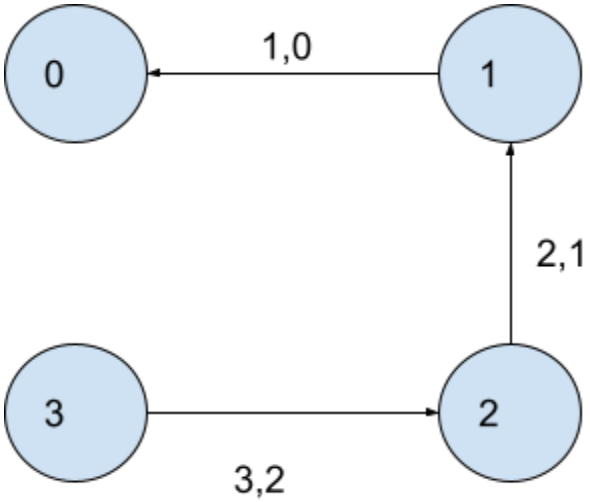
[[1,0], [2, 1], [3, 2]]

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] counter = 1

Q [2] counter = 2

Q [1] counter = 3

Q [0] counter = 4

Counter == no of nodes therefore these courses can be completed

If there had been a cycle then topological sort will result in counter being less than the number of nodes available and ,thus we will be able to detect the cycle this is how it detects completion of courses

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 counter variable initialized with 0
- Initialize queue with the elements having 0 indegree
- Traverse until the queue becomes empty :
    - Extract and pop the first element of the queue
    - Increase the counter variable
    - 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
- If the counter becomes equal to number of elements then return false as there is no cycle
- Return true otherwise meaning there is a cycle

Function Code

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

        // creating an empty queue
        // creating a counter variable having value 0
        queue<int> q;
```

```
        int counter = 0;

        // traversing through the indegree and adding those elements whose
indegree is 0 to queue
        for(int i=0;i<n;i++)
        {
            if(indegree[i]==0)
            {
                q.push(i);
            }
        }
        // traversing until the queue becomes empty
        while(!q.empty())
        {
            // extracting and popping the first node
            int node = q.front();
            q.pop();
            // increasing the counter
            counter++;
            // traverse for the adjacent node
            for(int i:adj[node])
            {
                // reducing the indegree
                indegree[i]--;
                // adding the 0 indegree elements to the queue
                if(indegree[i]==0)
                {
                    q.push(i);
                }
            }
        }
        // if a cycle encounters return true
        if(counter==n)return false;
        // return false if there is no cycle
        return true;
    }
      bool isPossible(int n,int p, vector<pair<int, int> >& prerequisites)
{
          // creating adjacency list
          vector<vector<int>> adj(n);
          for(auto it: prerequisites)
          {
              adj[it.first].push_back(it.second);
```

```
        }
        // returning true if there is not a cycle else returning false
        return !detect(n,adj);

    }
```

Time Complexity

O(V+E)