1462. Course Schedule IV

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `ai` first if you want to take course `bi`.

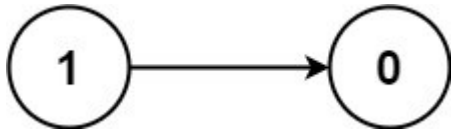
- For example, the pair `[0, 1]` indicates that you have to take course `0` before you can take course `1`.

Prerequisites can also be **indirect**. If course `a` is a prerequisite of course `b`, and course `b` is a prerequisite of course `c`, then course `a` is a prerequisite of course `c`.

You are also given an array `queries` where `queries[j] = [uj, vj]`. For the `jth` query, you should answer whether course `uj` is a prerequisite of course `vj` or not.

Return *a boolean array* `answer`*, where* `answer[j]` *is the answer to the* `jth` *query.*

Example 1:

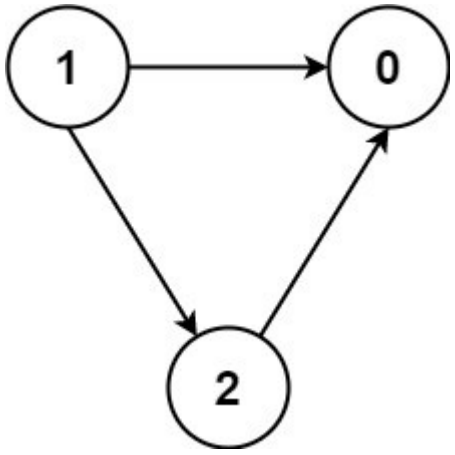



```
Input: numCourses = 2, prerequisites = [[1,0]], queries = [[0,1],[1,0]]
Output: [false,true]
Explanation: The pair [1, 0] indicates that you have to take course 1 before you can take course 0.
Course 0 is not a prerequisite of course 1, but the opposite is true.
```

Example 2:

```
Input: numCourses = 2, prerequisites = [], queries = [[1,0],[0,1]]
Output: [false,false]
Explanation: There are no prerequisites, and each course is independent.
```

Example 3:




```
Input: numCourses = 3, prerequisites = [[1,2],
[1,0],[2,0]], queries = [[1,0],[1,2]]
Output: [true,true]
```

Constraints:

- `2 <= numCourses <= 100`
- `0 <= prerequisites.length <= (numCourses * (numCourses - 1) / 2)`
- `prerequisites[i].length == 2`
- `0 <= ai, bi <= numCourses - 1`
- `ai != bi`
- All the pairs `[ai, bi]` are **unique**.
- The prerequisites graph has no cycles.
- `1 <= queries.length <=` 10^4
- `0 <= ui, vi <= numCourses - 1`
- `ui != vi`

Overview

We are given a directed graph representing course dependencies. The graph consists of `numCourses` nodes (denoted as `N` for simplicity) and `E` directed edges, where each edge is represented as a pair `(u, v)`. An edge `(u, v)` indicates that course `u` is a prerequisite for course `v`.

Additionally, we are given `Q` queries. Each query is a pair `(u, v)`, and the goal is to determine if course `u` is a prerequisite for course `v`. The answer to each query should be `true` if `u` is a prerequisite of `v`, and `false` otherwise.

Approach 3: Topological Sort - Kahn's Algorithm

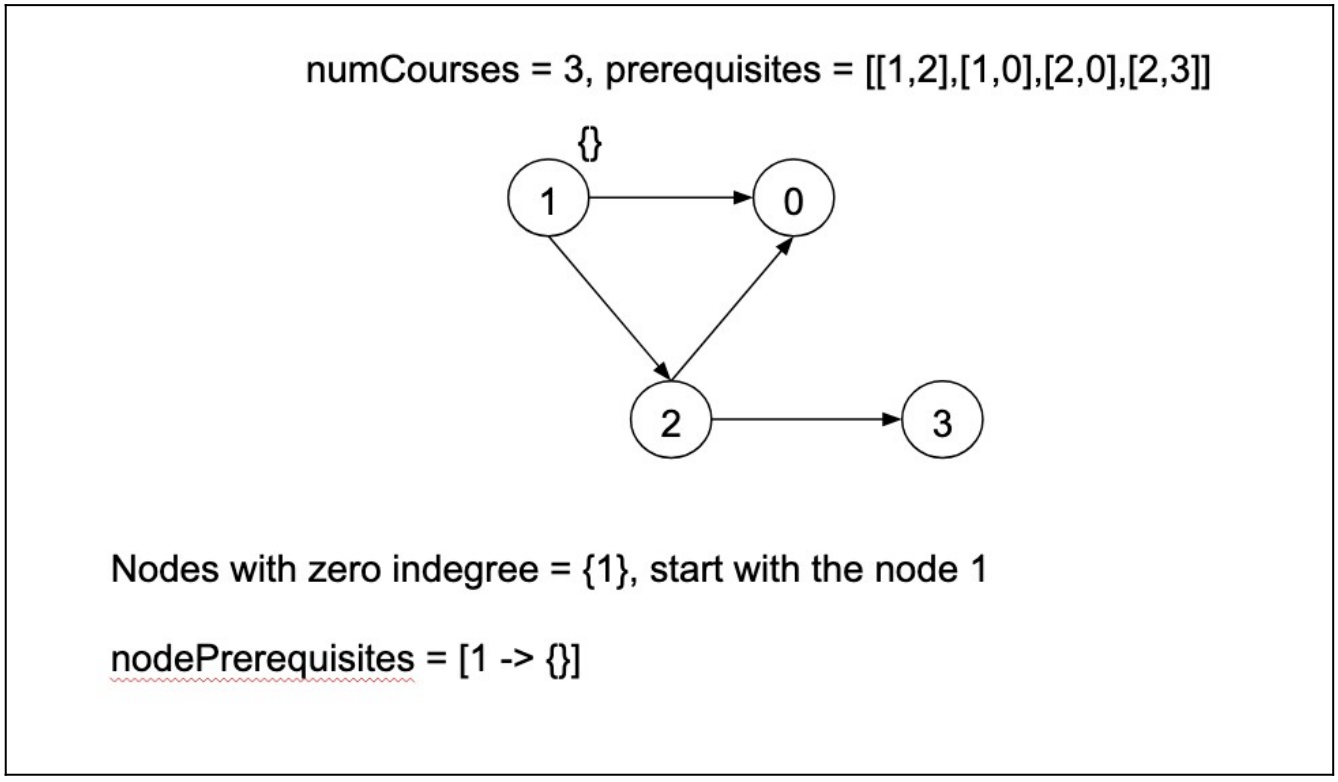
Intuition

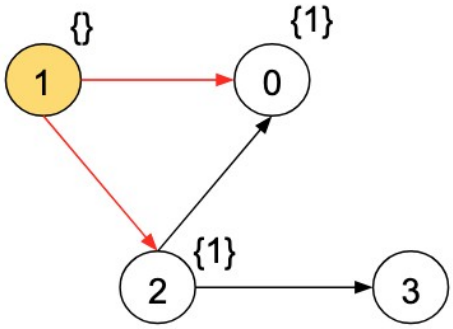
We need to find a way to process nodes in the correct order, ensuring that each node is processed only after its dependencies are handled. This is where ***topological sorting*** comes into play. Kahn's algorithm is a great fit for this task because it respects the dependencies of each node, ensuring nodes are only visited once their prerequisites are completed.

Now, to adapt Kahn's algorithm to our needs, we need to keep track of a node's prerequisites. Instead of just processing nodes in topological order, we'll modify the algorithm to maintain a list of dependencies for each node. As we move from node `u` to node `v`, we'll add all of `u`'s prerequisites to `v`'s prerequisites. This is important because it computes the transitive closure, meaning we're not just tracking immediate dependencies, but also indirect ones.

By the end of this process, each node will have a complete list of all nodes that must be visited before it. With this setup, when we need to answer a query `(u, v)`, all we have to do is check if `u` is in the list of prerequisites for `v`.

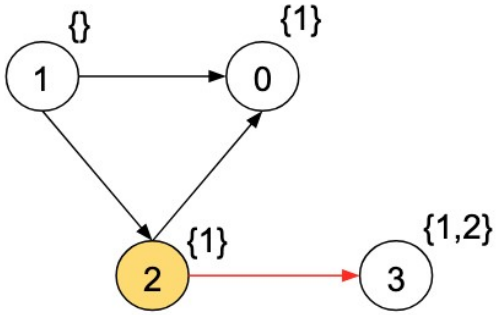
The general structure of Kahn's algorithm stays the same. We start by calculating the indegree of each node, which tells us how many nodes depend on it. Nodes with an indegree of zero are independent and can be processed first, so we enqueue them. Then, using a queue, we dequeue nodes, process their neighbors, update the prerequisite lists, and enqueue any neighbors whose indegree drops to zero. This continues until we've processed all nodes, ensuring the correct order of traversal.





From node 1, we will traverse to node 0 and 2, whose indegree will become zero. Add node 1 and it's prerequisites to both node's prerequisites.

nodePrerequisites = [1 -> {}, 0 -> {1}, 2 -> {1}]



From node 2, traverse to node 3 whose indegree will become zero. Add node 2 and it's prerequisites to node 3 prerequisite list.

nodePrerequisites = [1 -> {}, 0 -> {1}, 2 -> {1}, 3 -> {1, 2}]

1462. Course Schedule IV

```
/*
   Topological sort (compute prerequisites of each node)
   Time complexity: O(2n+n^2+n+n^3+qn)=O(3n+n^2+n^3+qn)
   Space complexity: O(n+2n^2)
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution {
   private:
      vvi graph;
      vi indegree;
      std::vector<std::unordered_set<int>> node_prerequisites;
   public:
      // Build the directed graph
      // <O(2n+n^2),O(n^2)>
      void build_graph(vvi& prerequisites,int n){
         graph.resize(n);
         indegree.resize(n,0);
         int m=prerequisites.size();
         for(int i=0;i<m;++i){
            int u=prerequisites[i][0];
            int v=prerequisites[i][1];
            graph[u].push_back(v);
            indegree[v]++;
         }
      }
```

```
    // Topological sort to compute the prerequisites of each node
    // <O(n+n^3),O(n+n^2)>
    void topo_sort(vvi& prerequisites,int n){
        node_prerequisites.resize(n);
        std::queue<int> q;
        for(int node=0;node<n;++node){
            if(indegree[node]==0) q.push(node);
        }

        while(!q.empty()){
            int cur_node=q.front();
            q.pop();

            for(auto& ne: graph[cur_node]){
                node_prerequisites[ne].insert(cur_node);
                for(auto& node: node_prerequisites[cur_node]) node_prerequisites[ne].insert(node);
                if(--indegree[ne]==0) q.push(ne);
            }
        }
    }

    std::vector<bool> checkIfPrerequisite(int numCourses,vvi& prerequisites,vvi& queries){
        build_graph(prerequisites,numCourses);

        topo_sort(prerequisites,numCourses);

        // Process queries
        // <O(q.n),O(1)>
        std::vector<bool> ans;
        for(auto& q: queries){
            int u=q[0],v=q[1];
            std::unordered_set<int> pre=node_prerequisites[v];
            ans.push_back(pre.find(u)!=pre.end());
        }

        return ans;
    }
};
```