Course Schedule II

There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

```
2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is `[0,1]`

```
4, [[1,0],[2,0],[3,1],[3,2]]
```

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is `[0,1,2,3]`. Another correct ordering is `[0,2,1,3]`.

Note:
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

click to show more hints.

Hints:

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Solution 1

This question asks for an order in which prerequisite courses must be taken first. This prerequisite relationship reminds one of directed graphs. Then, the problem reduces to find a topological sort order of the courses, which would be a DAG if it has a valid order.

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    int[] incLinkCounts = new int[numCourses];
    List<List<Integer>> adjs = new ArrayList<>(numCourses);
    initialiseGraph(incLinkCounts, adjs, prerequisites);
    //return solveByBFS(incLinkCounts, adjs);
    return solveByDFS(adjs);
}
```

The first step is to transform it into a directed graph. Since it is likely to be sparse,we use adjacency list graph data structure. 1 -> 2 means 1 must be taken before 2.

```
private void initialiseGraph(int[] incLinkCounts, List<List<Integer>> adjs, int[]
[] prerequisites){
    int n = incLinkCounts.length;
    while (n-- > 0) adjs.add(new ArrayList<>());
    for (int[] edge : prerequisites) {
        incLinkCounts[edge[0]]++;
        adjs.get(edge[1]).add(edge[0]);
    }
}
```

How can we obtain a topological sort order of a DAG?

We observe that if a node has incoming edges, it has prerequisites. Therefore, the first few in the order must be those with no prerequisites, i.e. no incoming edges. Any non-empty DAG must have at least one node without incoming links. You can draw a small graph to convince yourself. If we visit these few and remove all edges attached to them, we are left with a smaller DAG, which is the same problem. This will then give our BFS solution.

```
private int[] solveByBFS(int[] incLinkCounts, List<List<Integer>> adjs){
    int[] order = new int[incLinkCounts.length];
    Queue<Integer> toVisit = new ArrayDeque<>();
    for (int i = 0; i < incLinkCounts.length; i++) {
        if (incLinkCounts[i] == 0) toVisit.offer(i);
    }
    int visited = 0;
    while (!toVisit.isEmpty()) {
        int from = toVisit.poll();
        order[visited++] = from;
        for (int to : adjs.get(from)) {
            incLinkCounts[to]--;
            if (incLinkCounts[to] == 0) toVisit.offer(to);
        }
    }
    return visited == incLinkCounts.length ? order : new int[0];
}
```

Another way to think about it is the last few in the order must be those which are not prerequisites of other courses. Thinking it recursively means if one node has unvisited child node, you should visit them first before you put this node down in the final order array. This sounds like the post-order of a DFS. Since we are putting nodes down in the reverse order, we should reverse it back to correct ordering or use a stack.

```java
private int[] solveByDFS(List<List<Integer>> adjs) {
    BitSet hasCycle = new BitSet(1);
    BitSet visited = new BitSet(adjs.size());
    BitSet onStack = new BitSet(adjs.size());
    Deque<Integer> order = new ArrayDeque<>();
    for (int i = adjs.size() - 1; i >= 0; i--) {
        if (visited.get(i) == false && hasOrder(i, adjs, visited, onStack, order)
== false) return new int[0];
    }
    int[] orderArray = new int[adjs.size()];
    for (int i = 0; !order.isEmpty(); i++) orderArray[i] = order.pop();
    return orderArray;
}

private boolean hasOrder(int from, List<List<Integer>> adjs, BitSet visited, BitS
et onStack, Deque<Integer> order) {
    visited.set(from);
    onStack.set(from);
    for (int to : adjs.get(from)) {
        if (visited.get(to) == false) {
            if (hasOrder(to, adjs, visited, onStack, order) == false) return fals
e;
        } else if (onStack.get(to) == true) {
            return false;
        }
    }
    onStack.clear(from);
    order.push(from);
    return true;
}
```

written by lx223 original link here

Solution 2

Well, this problem is spiritually similar to to Course Schedule. You only need to store the nodes in the order you visit into a vector during BFS or DFS. Well, for DFS, a final reversal is required.

BFS

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)
{
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<int> degrees = compute_indegree(graph);
        queue<int> zeros;
        for (int i = 0; i < numCourses; i++)
            if (!degrees[i]) zeros.push(i);
        vector<int> toposort;
        for (int i = 0; i < numCourses; i++) {
            if (zeros.empty()) return {};
            int zero = zeros.front();
            zeros.pop();
            toposort.push_back(zero);
            for (int neigh : graph[zero]) {
                if (!--degrees[neigh])
                    zeros.push(neigh);
            }
        }
        return toposort;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    vector<int> compute_indegree(vector<unordered_set<int>>& graph) {
        vector<int> degrees(graph.size(), 0);
        for (auto neighbors : graph)
            for (int neigh : neighbors)
                degrees[neigh]++;
        return degrees;
    }
};
```

DFS

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)
{
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<int> toposort;
        vector<bool> onpath(numCourses, false), visited(numCourses, false);
        for (int i = 0; i < numCourses; i++)
            if (!visited[i] && dfs(graph, i, onpath, visited, toposort))
                return {};
        reverse(toposort.begin(), toposort.end());
        return toposort;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    bool dfs(vector<unordered_set<int>>& graph, int node, vector<bool>& onpath, v
ector<bool>& visited, vector<int>& toposort) {
        if (visited[node]) return false;
        onpath[node] = visited[node] = true;
        for (int neigh : graph[node])
            if (onpath[neigh] || dfs(graph, neigh, onpath, visited, toposort))
                return true;
        toposort.push_back(node);
        return onpath[node] = false;
    }
};
```

written by jianchao.li.fighter original link here

Solution 3

```
public class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<List<Integer>> adj = new ArrayList<>(numCourses);
        for (int i = 0; i < numCourses; i++) adj.add(i, new ArrayList<>());
        for (int i = 0; i < prerequisites.length; i++) adj.get(prerequisites[i][1
]).add(prerequisites[i][0]);
        boolean[] visited = new boolean[numCourses];
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < numCourses; i++) {
            if (!topologicalSort(adj, i, stack, visited, new boolean[numCourses])
) return new int[0];
        }
        int i = 0;
        int[] result = new int[numCourses];
        while (!stack.isEmpty()) {
            result[i++] = stack.pop();
        }
        return result;
    }

    private boolean topologicalSort(List<List<Integer>> adj, int v, Stack<Integer
> stack, boolean[] visited, boolean[] isLoop) {
        if (visited[v]) return true;
        if (isLoop[v]) return false;
        isLoop[v] = true;
        for (Integer u : adj.get(v)) {
            if (!topologicalSort(adj, u, stack, visited, isLoop)) return false;
        }
        visited[v] = true;
        stack.push(v);
        return true;
    }
}
```

written by baibing original link here