Description

210. 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].

Idea Report

The problem is equivalent of find the topological order of a directed graph, where each course is a vertex and if A's prerequisite course is B then B->A is an edge. To get the topological order we can do both BFS (prefered) and DFS.

The first step is to find all the courses that has no prerequisites, these courses are the starting point (with 0 indegree). After finishing all the staring point a,b,c, there must be some courses d,e,f which prerequisites are these starting points a,b,c, so then we can take those classes because we have already finished their prerequisites. And starting from there we can take next step. So we can draw something like course X point to course Y then point to course Z. So our starting point are the nodes with indegree equals to zeros which are the courses with no prerequisites. We finish X first, when we finish X we decrese Y's indegree by 1 because Y's prerequisite course X has been finished. Now Y's indegree changes to 0, so we can now take Y and decrease Z's indegree by 1. So then we take Z. Finally if the number of courses taken equals the total number of courses n, we get the topological order and return, otherwise there is a loop in the graph so that we are not able to finish all the courses, so we return a empty array.

Code

```
public class Solution {
    // BFS AC
    public int[] findOrder(int n, int[][] pre) {
        List<Integer>[] adj = new List[n];
        for (int i = 0; i < n; i++) {
            adj[i] = new ArrayList<>();
        }

        int[] indegree = new int[n];
        for (int[] p : pre) {
            indegree[p[0]]++;
            adj[p[1]].add(p[0]);
        }

        Deque<Integer> q = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                q.offer(i);
            }
        }

        int[] res = new int[n];
        int index = 0;
        while (!q.isEmpty()) {
            int cur = q.poll();
            res[index++] = cur;
            for (int nei : adj[cur]) {
                indegree[nei]--;
                if (indegree[nei] == 0) {
                    q.offer(nei);
                }
            }
```

```
        }

        return index == n ? res : new int[0];
    }
}
```

Summary

- Node-to-node representation can be convertd to a directed graph.
- Topological sorting.
- List[] adj = new List[n]; is the new thing that I learned.
- The time complexity would be the number of Vertex + number of Edges
- The space complexity would be the number of Vertex, actually is the queue's largest size while traversing.

Follow Up

The DFS can also achieve topological sorting, but not really very intuitive. Basically for each node (course), we do a DFS and see if there is circle in the graph. If there is, then those courses cannot be finished, we return an empty array, otherwise we can return the order. The deepest node is the last node we want to take, so we can fill the output order array from the last index.

Code

```
public class Solution {
    // DFS AC
    public int[] findOrder(int n, int[][] pre) {
        List<Integer>[] adj = new List[n];
        for (int i = 0; i < n; i++) {
            adj[i] = new ArrayList<>();
        }

        for (int[] p : pre) {
            adj[p[1]].add(p[0]);
        }

        int[] res = new int[n];
        int[] visited = new int[n];
        int[] index = new int[1];
```

```
        index[0] = n - 1;

        for (int i = 0; i < n; i++) {
            if (visited[i] == -1) {
                continue;
            }
            if (findCircle(res, index, adj, visited, i)) {
                return new int[0];
            }
        }


        return index[0] == -1 ? res : new int[0];
    }

    private boolean findCircle(int[] res, int[] index, List<Integer>[] adj,
                               int[] visited, int cur) {
        if (visited[cur] == 1) {
            return true;
        } else if (visited[cur] == -1) {
            return false;
        }

        visited[cur] = 1;
        for (int next : adj[cur]) {
            if (visited[next] == -1) {
                continue;
            }

            if (findCircle(res, index, adj, visited, next)) {
                return true;
            }
        }

        visited[cur] = -1;
        res[index[0]--] = cur;
        return false;
    }
}
```

C++ implementation

```
class Solution {
public:
    // BFS AC
    vector<int> findOrder(int n, vector<pair<int, int>>& pre) {
        int indegree[n] = {0};
```

```
        unordered_map<int, vector<int>> map;
        for (auto p : pre) {
            indegree[p.first]++;
            map[p.second].push_back(p.first);
        }

        deque<int> q;
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                q.push_back(i);
            }
        }

        vector<int> res;
        while (!q.empty()) {
            int cur = q.front();
            q.pop_front();
            res.push_back(cur);
            for (auto next : map[cur]) {
                indegree[next]--;
                if (indegree[next] == 0) {
                    q.push_back(next);
                }
            }
        }
        return res.size() == n ? res : vector<int>();
    }
};
```

```
class Solution {
public:
    // DFS AC
    vector<int> findOrder(int n, vector<pair<int, int>>& pre) {
        unordered_map<int, vector<int>> map;
        for (auto p : pre) {
            map[p.second].push_back(p.first);
        }

        vector<int> res(n);
        int visited[n] = {0};
        int index = n - 1;
        for (int i = 0; i < n; i++) {
            if (visited[i] == -1) {
                continue;
            }
            if (hasCircle(map, res, index, i, visited)) {
                return vector<int>();
            }
        }
```

```
        return res;
    }

    bool hasCircle(unordered_map<int, vector<int>>& map, vector<int>& res,
                   int& index, int cur, int visited[]) {
        if (visited[cur] == -1) {
            return false;
        }

        if (visited[cur] == 1) {
            return true;
        }

        visited[cur] = 1;
        for (auto next : map[cur]) {
            if (hasCircle(map, res, index, next, visited)) {
                return true;
            }
        }

        visited[cur] = -1;
        res[index--] = cur;
        return false;
    }
}
```