<https://leetcode.com/problems/course-schedule-ii/>

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 bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

**Example 1:**

Input: numCourses = 2, prerequisites = [[1,0]]

Output: [0,1]

Explanation: 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].

**Example 2:**

Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

Output: [0,2,1,3]

Explanation: 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].

**Example 3:**

Input: numCourses = 1, prerequisites = []

Output: [0]

**Constraints:**

1 <= numCourses <= 2000

0 <= prerequisites.length <= numCourses \* (numCourses - 1)

prerequisites[i].length == 2

0 <= ai, bi < numCourses

ai != bi

All the pairs [ai, bi] are **distinct**.

**Attempt 1: 2022-11-20**

**Solution 1:  Detect Cycle in a Directed Graph using BFS [Topological Sort ] (10min)**

class Solution {

    public int[] findOrder(int numCourses, int[][] prerequisites) {

        // Build graph

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < numCourses; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(int[] pre : prerequisites) {

            graph.get(pre[1]).add(pre[0]);

        }

        // Get indegree based on graph

        Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

        for(int node : graph.keySet()) {

            indegree.put(node, 0);

        }

        for(int node : graph.keySet()) {

            for(int neighbour : graph.get(node)) {

                indegree.put(neighbour, indegree.get(neighbour) + 1);

            }

        }

        // Topological Sort

        int[] result = new int[numCourses];

        Queue<Integer> q = new LinkedList<Integer>();

        for(int node : indegree.keySet()) {

            if(indegree.get(node) == 0) {

                q.offer(node);

            }

        }

        // 'count' for recording number of courses can take

        int count = 0;

        while(!q.isEmpty()) {

            int node = q.poll();

            result[count] = node;

            count++;

            for(int neighbour : graph.get(node)) {

                indegree.put(neighbour, indegree.get(neighbour) - 1);

                if(indegree.get(neighbour) == 0) {

                    q.offer(neighbour);

                }

            }

        }

        if(count == numCourses) {

            return result;

        } else {

            return new int[]{};

        }

    }

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number

of times that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any

one iteration of the outer loop, it will only be executed once for each successor of each member,

which means that the total number of times that it will be executed is the total number of

successors of all the members -- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information

stored for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing

the successors for each member; once again, the total number of successors is the number of

relations, so O(m).

**Solution 2:  Detect Cycle in a Directed Graph using DFS [Backtracking] (10min)**

class Solution {

    public int[] findOrder(int numCourses, int[][] prerequisites) {

        List<Integer> tmp = new ArrayList<Integer>();

        boolean[] visited = new boolean[numCourses];

        boolean[] recursionStack = new boolean[numCourses];

        // Build graph

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < numCourses; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(int[] pre : prerequisites) {

            graph.get(pre[1]).add(pre[0]);

        }

        // DFS to find cycle based on each node

        for(int i = 0; i < numCourses; i++) {

            // Additional check 'visited' or not to promote

            // efficiency on the fly since 'visited' will be

            // update during DFS

            if(!visited[i]) {

                // If we have cycle then not able to finish

                if(hasCycle(i, graph, visited, recursionStack, tmp)) {

                    return new int[]{};

                }

            }

        }

        int[] result = new int[numCourses];

        for(int i = 0; i < numCourses; i++) {

            result[i] = tmp.get(i);

        }

        return result;

    }

    private boolean hasCycle(int course, Map<Integer, List<Integer>> graph, boolean[] visited, boolean[] recursionStack, List<Integer> tmp) {

        // If course detect on same path again then find cycle

        if(recursionStack[course]) {

            return true;

        }

        if(visited[course]) {

            return false;

        }

        // Don't add 'course' before 'no cycle' confirmed

        // Test out:

        // Input: 4, [[1,0],[2,0],[3,1],[3,2]]

        // Output: [0,1,3,2]

        // Expected: [0,2,1,3]

        //tmp.add(course);

        visited[course] = true;

        // Backtrack the current path since switch to other path

        recursionStack[course] = true;

        for(int neighbour : graph.get(course)) {

            if(hasCycle(neighbour, graph, visited, recursionStack, tmp)) {

                return true;

            }

        }

        recursionStack[course] = false;

        // The only difference than L207.Course Schedule which record path

        // 1. Must put 'course' into path reversely, always insert at index = 0

        // 2. Only add 'course' after 'no cycle' confirmed

        tmp.add(0, course);

        return false;

    }

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times

that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration

of the outer loop, it will only be executed once for each successor of each member, which means that

the total number of times that it will be executed is the total number of successors of all the members

-- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored

for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors

for each member; once again, the total number of successors is the number of relations, so O(m).

**Refer to**

<https://leetcode.com/problems/course-schedule-ii/discuss/59317/Two-AC-solution-in-Java-using-BFS-and-DFS-with-explanation/60618>

public class Solution {

    public int[] findOrder(int numCourses, int[][] prerequisites) {

        //prepare

        List<List<Integer>> graph = new ArrayList<>();

        for(int i = 0; i < numCourses; i++){

            graph.add(new ArrayList<>());

        }

        for(int[] pair : prerequisites){

            int prev = pair[1];

            int next = pair[0];

            graph.get(prev).add(next);

        }

        Map<Integer, Integer> visited = new HashMap<>();

        //initail visited

        for(int i = 0; i < numCourses; i++){

            visited.put(i, 0);//0 -> unvisited, 1 -> visiting, 2 -> visited

        }

        List<Integer> res = new ArrayList<>();

        for(int i = 0; i < numCourses; i++){

            if(!topoSort(res, graph, visited, i)) return new int[0];

        }

        int[] result = new int[numCourses];

        for(int i = 0; i < numCourses; i++){

            result[i] = res.get(numCourses - i - 1);

        }

        return result;

    }

    //the return value of this function only contains the ifCycle info and does not interfere dfs process. if there is Cycle, then return false

    private boolean topoSort(List<Integer> res, List<List<Integer>> graph, Map<Integer, Integer> visited, int i){

        int visit = visited.get(i);

        if(visit == 2){//when visit = 2, which means the subtree whose root is i has been dfs traversed and all the nodes in subtree has been put in the result(if we request), so we do not need to traverse it again

            return true;

        }if(visit == 1){

            return false;

        }

        visited.put(i, 1);

        for(int j : graph.get(i)){

            if(!topoSort(res, graph, visited, j)) return false;

        }

        visited.put(i, 2);

        res.add(i);//the only difference with traversing a graph

        return true;

    }

}

**Refer to**

[L207.P17.2.Course Schedule (Ref.L210,L261)](note://C74CA92E9FAA407FAC5C404EC10B0262)

[L261.Lint178.Graph Valid Tree (Ref.L841)](note://2DB801CF2BAB4DDE9E74C819757178B5)

[Directed and Undirected Graph Cycle Detection in DFS and BFS](note://8D51E6EB87834766A8F4688C191B70AC)