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**.

Solution

This is a very common problem that some of us might face during college. We might want to take up a certain set of courses that interest us. However, as we all know, most of the courses do tend to have a lot of prerequisites associated with them. Some of these would be hard requirements whereas others would be simply suggested prerequisites which you may or may not take. However, for us to be able to have an all round learning experience, we should follow the suggested set of prerequisites. How does one decide what order of courses they should follow so as not to miss out on any subjects?

As mentioned in the problem statement, such a problem is a natural fit for graph based algorithms and we can easily model the elements in the problem statement as a graph. First of all, let's look at the graphical representation of the problem and it's components and then we will move onto the solutions.

We can represent the information provided in the question in the form of a graph.

* Let G(V, E)*G*(*V*,*E*) represent a directed, unweighted graph.
* Each course would represent a vertex in the graph.
* The edges are modeled after the prerequisite relationship between courses. So, we are given, that a pair such as [a, b][*a*,*b*] in the question means the course b is a prerequisite for the course a. This can be represented as a directed edge b ➔ a in the graph.
* The graph is a cyclic graph because there is a possibility of a cycle in the graph. If the graph would be acyclic, then an ordering of subjects as required in the question would always be possible. Since it's mentioned that such an ordering may not always be possible, that means we have a cyclic graph.

Let's look at a sample graph representing a set of courses where such an ordering is possible and one where such an ordering is not possible. It will be easier to explain the approaches once we look at two sample graphs.

Diagram

Description automatically generated

For the sample graph shown above, one of the possible ordering of courses is: C6 ➔ C4 ➔ C1 ➔ C5 ➔ C2 ➔ C3 and another possible ordering of subjects is C6 ➔ C4 ➔ C5 ➔ C1 ➔ C2 ➔ C3. Now let's look at a graph where no such ordering of courses is possible.

Diagram

Description automatically generated

Note that the edges that have changed from the previous graph have been highlighted in red.

Clearly, the presence of a cycle in the graph shows us that a proper ordering of prerequisites is not possible at all. Intuitively, it is not possible to have e.g. two subjects S1 and S2 prerequisites of each other. Similar ideology applies to a larger cycle in the graph like we have above.

Such an ordering of subjects is referred to as a Topological Sorted Order and this is a common algorithmic problem in the graph domain. There are two approaches that we will be looking at in this article to solve this problem.

Approach 1: Using Depth First Search

**Intuition**

Suppose we are at a node in our graph during the depth first traversal. Let's call this node A.

The way DFS would work is that we would consider all possible paths stemming from A before finishing up the recursion for A and moving onto other nodes. All the nodes in the paths stemming from the node A would have A as an ancestor. The way this fits in our problem is, all the courses in the paths stemming from the course A would have A as a prerequisite.

Now we know how to get all the courses that have a particular course as a prerequisite. If a valid ordering of courses is possible, the course A would come before all the other set of courses that have it as a prerequisite. This idea for solving the problem can be explored using depth first search. Let's look at the pseudo-code before looking at the formal algorithm.

➔ let S be a stack of courses

➔ function dfs(node)

➔ for each neighbor in adjacency list of node

➔ dfs(neighbor)

➔ add node to S

Let's now look at the formal algorithm based on this idea.

**Algorithm**

1. Initialize a stack S that will contain the topologically sorted order of the courses in our graph.
2. Construct the adjacency list using the edge pairs given in the input. An important thing to note about the input for the problem is that a pair such as [a, b] represents that the course b needs to be taken in order to do the course a. This implies an edge of the form b ➔ a. Please take note of this when implementing the algorithm.
3. For each of the nodes in our graph, we will run a depth first search in case that node was not already visited in some other node's DFS traversal.
4. Suppose we are executing the depth first search for a node N. We will recursively traverse all of the neighbors of node N which have not been processed before.
5. Once the processing of all the neighbors is done, we will add the node N to the stack. We are making use of a stack to simulate the ordering we need. When we add the node N to the stack, all the nodes that require the node N as a prerequisites (among others) will already be in the stack.
6. Once all the nodes have been processed, we will simply return the nodes as they are present in the stack from top to bottom.

Let's look at an animated dry run of the algorithm on a sample graph before moving onto the formal implementations.

An important thing to note about topologically sorted order is that there won't be just one ordering of nodes (courses). There can be multiple. For e.g. in the above graph, we could have processed the node "D" before we did "B" and hence have a different ordering.

class Solution {

static int WHITE = 1;

static int GRAY = 2;

static int BLACK = 3;

boolean isPossible;

Map<Integer, Integer> color;

Map<Integer, List<Integer>> adjList;

List<Integer> topologicalOrder;

private void init(int numCourses) {

this.isPossible = true;

this.color = new HashMap<Integer, Integer>();

this.adjList = new HashMap<Integer, List<Integer>>();

this.topologicalOrder = new ArrayList<Integer>();

// By default all vertces are WHITE

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

this.color.put(i, WHITE);

}

}

private void dfs(int node) {

// Don't recurse further if we found a cycle already

if (!this.isPossible) {

return;

}

// Start the recursion

this.color.put(node, GRAY);

// Traverse on neighboring vertices

for (Integer neighbor : this.adjList.getOrDefault(node, new ArrayList<Integer>())) {

if (this.color.get(neighbor) == WHITE) {

this.dfs(neighbor);

} else if (this.color.get(neighbor) == GRAY) {

// An edge to a GRAY vertex represents a cycle

this.isPossible = false;

}

}

// Recursion ends. We mark it as black

this.color.put(node, BLACK);

this.topologicalOrder.add(node);

}

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

this.init(numCourses);

// Create the adjacency list representation of the graph

for (int i = 0; i < prerequisites.length; i++) {

int dest = prerequisites[i][0];

int src = prerequisites[i][1];

List<Integer> lst = adjList.getOrDefault(src, new ArrayList<Integer>());

lst.add(dest);

adjList.put(src, lst);

}

// If the node is unprocessed, then call dfs on it.

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

if (this.color.get(i) == WHITE) {

this.dfs(i);

}

}

int[] order;

if (this.isPossible) {

order = new int[numCourses];

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

order[i] = this.topologicalOrder.get(numCourses - i - 1);

}

} else {

order = new int[0];

}

return order;

}

}

**Complexity Analysis**

* Time Complexity: O(V + E)*O*(*V*+*E*) where V*V* represents the number of vertices and E*E* represents the number of edges. Essentially we iterate through each node and each vertex in the graph once and only once.
* Space Complexity: O(V + E)*O*(*V*+*E*).
  + We use the adjacency list to represent our graph initially. The space occupied is defined by the number of edges because for each node as the key, we have all its adjacent nodes in the form of a list as the value. Hence, O(E)*O*(*E*)
  + Additionally, we apply recursion in our algorithm, which in worst case will incur O(E)*O*(*E*) extra space in the function call stack.
  + To sum up, the overall space complexity is O(V + E)*O*(*V*+*E*).

Approach 2: Using Node Indegree

**Intuition**

This approach is much easier to think about intuitively as will be clear from the following point/fact about topological ordering.

The first node in the topological ordering will be the node that doesn't have any incoming edges. Essentially, any node that has an in-degree of 0 can start the topologically sorted order. If there are multiple such nodes, their relative order doesn't matter and they can appear in any order.

Our current algorithm is based on this idea. We first process all the nodes/course with 0 in-degree implying no prerequisite courses required. If we remove all these courses from the graph, along with their outgoing edges, we can find out the courses/nodes that should be processed next. These would again be the nodes with 0 in-degree. We can continuously do this until all the courses have been accounted for.

**Algorithm**

1. Initialize a queue, Q to keep a track of all the nodes in the graph with 0 in-degree.
2. Iterate over all the edges in the input and create an adjacency list and also a map of node v/s in-degree.
3. Add all the nodes with 0 in-degree to Q.
4. The following steps are to be done until the Q becomes empty.
   1. Pop a node from the Q. Let's call this node, N.
   2. For all the neighbors of this node, N, reduce their in-degree by 1. If any of the nodes' in-degree reaches 0, add it to the Q.
   3. Add the node N to the list maintaining topologically sorted order.
   4. Continue from step 4.1.

Let us now look at an animation depicting this algorithm and then we will get to the implementations.

An important thing to note here is, using a queue is not a hard requirement for this algorithm. We can make use of a stack. That however, will give us a different ordering than what we might get from the queue because of the difference in access patterns between the two data-structures.

class Solution {

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

boolean isPossible = true;

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

int[] indegree = new int[numCourses];

int[] topologicalOrder = new int[numCourses];

// Create the adjacency list representation of the graph

for (int i = 0; i < prerequisites.length; i++) {

int dest = prerequisites[i][0];

int src = prerequisites[i][1];

List<Integer> lst = adjList.getOrDefault(src, new ArrayList<Integer>());

lst.add(dest);

adjList.put(src, lst);

// Record in-degree of each vertex

indegree[dest] += 1;

}

// Add all vertices with 0 in-degree to the queue

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

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

if (indegree[i] == 0) {

q.add(i);

}

}

int i = 0;

// Process until the Q becomes empty

while (!q.isEmpty()) {

int node = q.remove();

topologicalOrder[i++] = node;

// Reduce the in-degree of each neighbor by 1

if (adjList.containsKey(node)) {

for (Integer neighbor : adjList.get(node)) {

indegree[neighbor]--;

// If in-degree of a neighbor becomes 0, add it to the Q

if (indegree[neighbor] == 0) {

q.add(neighbor);

}

}

}

}

// Check to see if topological sort is possible or not.

if (i == numCourses) {

return topologicalOrder;

}

return new int[0];

}

}

**Complexity Analysis**

* Time Complexity: O(V + E)*O*(*V*+*E*) where V*V* represents the number of vertices and E*E* represents the number of edges. We pop each node exactly once from the zero in-degree queue and that gives us V*V*. Also, for each vertex, we iterate over its adjacency list and in totality, we iterate over all the edges in the graph which gives us E*E*. Hence, O(V + E)*O*(*V*+*E*)
* Space Complexity: O(V + E)*O*(*V*+*E*). We use an intermediate queue data structure to keep all the nodes with 0 in-degree. In the worst case, there won't be any prerequisite relationship and the queue will contain all the vertices initially since all of them will have 0 in-degree. That gives us O(V)*O*(*V*). Additionally, we also use the adjacency list to represent our graph initially. The space occupied is defined by the number of edges because for each node as the key, we have all its adjacent nodes in the form of a list as the value. Hence, O(E)*O*(*E*). So, the overall space complexity is O(V + E)*O*(*V*+*E*).