Link to the problem

https://leetcode.com/problems/course-schedule-ii/
Note: This isn't an exact match of this problem, the only difference is that the LeetCode problem requires us to return "any" valid answer while this problem requires "all" valid answers.

The underlying problem

This problem consists of two layers, we're going to dive into the lower one first here.
Problem: We want to find all possible unique orderings between the following three elements, taking every element exactly once in every ordering.

We have no other constraints on the order between them. What should we do?
We shall move step by step, and consider all the available options at every step, and try taking each one of them once.

Step 1:
What we have: nothing yet
Our options: 1 , 2 , 3
Decision: Take 1
Result: 1

Step 2:
What we have: 1
Our options: 2 , 3
Decision: Take 2
Result: 1 , 2

Step 3:
What we have: 1 , 2
Our options: 3
Decision: Take 3
Result: 1 , 2 , 3
We have exhausted all of our options at this point. We have reached one possible answer at this point which is the unique order 1 , 2 , 3.

We shall now go to step 4, which will continue from the last step that had more unexplored options, that is step 2.

Step 4:
What we have: 1
Our options: 2 , 3
Decision: Take 3 (This time we took 3 instead of 2)
Result: 1 , 3

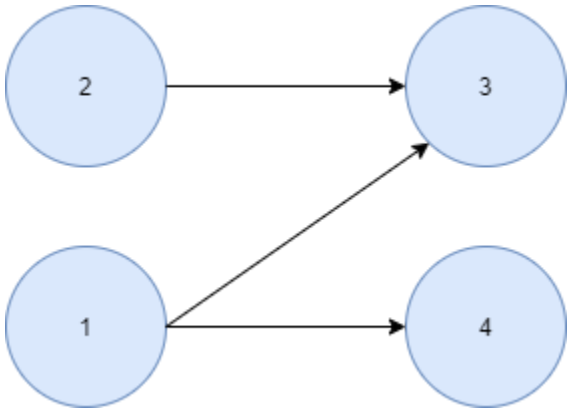
Step 5:
What we have: 1 , 3
Our options: 2
Decision: Take 2
Result: 1 , 3 , 2
We have again exhausted all of our options at this point. We have reached one possible answer at this point which is the unique order 1 , 3 , 2.
And we keep doing this until we have exhausted all of our options at every step.

In mathematics, this is called permutations. There is a mathematical formula to find out how many final results there can be to a problem like this. The formula is:

Factorial(N), where N is the total number of elements and the Factorial of N is equal to:
1 * 2 * 3 * … * (N-1) * N

The main problem: Finding all course orders

In reality, this problem is a special case of the problem explained above, the only difference is there are additional constraints that control which nodes are available at every step as “options”. The constraint that decides whether a node is an “option” or not at a certain step is whether it has 0 in-degrees or not. Once we choose a node as a “decision”, we want to decrease the in-degree of all its neighboring nodes by 1. Once we go back on a “decision”, we want to increase the in-degree of all its neighboring nodes by 1.



Step 1:
What we have: nothing yet
Our options: 1 , 2 (The only two nodes that match our constraint)
Decision: Take 1 (And decrease the in-degrees of nodes (3, 4) by 1)
Result: 1

Step 2:
What we have: 1
Our options: 2 , 4
Decision: Take 4
Result: 1 , 4
And so on.

Time and space complexities

Time complexity

In the worst-case scenario, this problem will be the same as the underlying problem where we have no constraints at all. This is because the fewer constraints you have, the more options/ordering you'll end up with. This gives us a total of Factorial(V) orderings. It is true that this case only occurs when there are no edges between the nodes, but we must still take the worst case for the edge traversal. The worst case is we pay E at every node "decision". This gives us the final complexity of O(Factorial(V) * E)

Space complexity

The space complexity is O(V). This comes from two sources:

1. The call stack
    a. This might grow to O(V) in the worst case, it cannot grow more than the total number of "decisions" that we can make, which is "V".
2. The array that keeps track of our decisions
    a. This might grow to O(V) in the worst case, it cannot grow more than the total number of "decisions" that we can make, which is "V".