**Algorithm for flight Path trace**

(This is implemented)

`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.

Algorithm is based on this idea.

We first process all the nodes/flights with 0 in-degree implying no source flight. If we remove all these flights from the graph, along with their outgoing edges, we can find out the nodes that should be processed next. These would again be the nodes with 0 in-degree. We can continuously do this until all the flights 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.

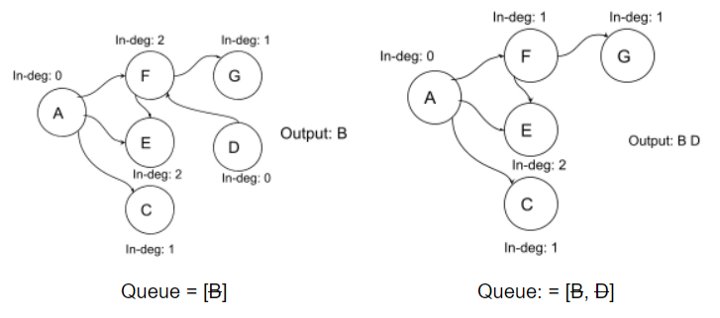
3. Add the node N to the list maintaining topologically sorted order.

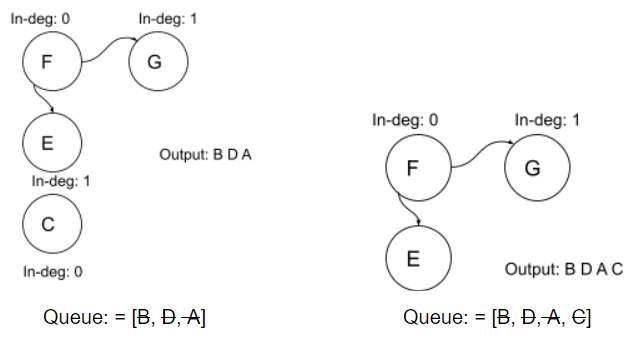
4. Continue from step 4.1.

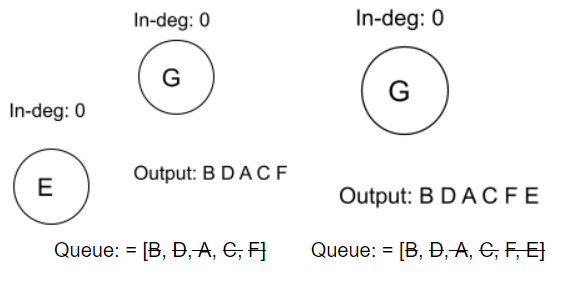
**Complexity Analysis**

* Time Complexity: *O*(*V*+*E*) where *V* represents the number of vertices and *E* represents the number of edges. We pop each node exactly once from the zero in-degree queue and that gives us *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*. Hence, *O*(*V*+*E*)
* Space Complexity: *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*). 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*). So, the overall space complexity is *O*(*V*+*E*).

Algorithm illustration-



****

****

**Alternate Algorithm using 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.

➔ let S be a stack of flight paths

➔ 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

Initialize a stack S that will contain the topologically sorted order of the s in our graph.

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.

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.

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.

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.

Once all the nodes have been processed, we will simply return the nodes as they are present in the stack from top to bottom.

**Complexity Analysis**

* Time Complexity: *O*(*V*+*E*) where *V* represents the number of vertices and *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*).
  + 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*)
  + Additionally, we apply recursion in our algorithm, which in worst case will incur *O*(*E*) extra space in the function call stack.
  + To sum up, the overall space complexity is *O*(*V*+*E*).