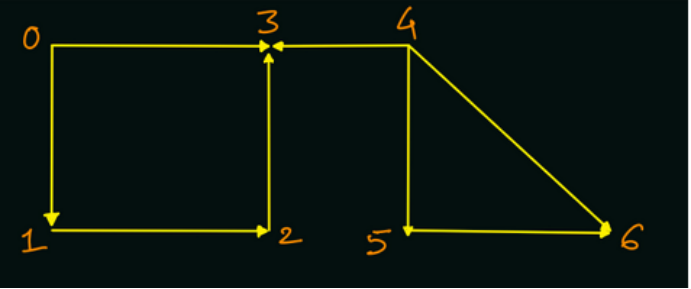
**1. Understanding Problem:**

You are given a directed acyclic graph. The vertices represent tasks and edges represent dependencies between tasks. You are required to find and print the order in which tasks could be done. The task that should be done at last should be printed first and the task which should be done first should be printed last. This kind of ordering can be achieved by the topological sort of the graph. Topological sort: A permutation of vertices for a directed acyclic graph is called topological sort if for all directed edges uv, u appears before v in the graph. Topological sorting for a graph is not possible if the graph is undirected or if the graph has a cycle. Hence, it must be Directed Acyclic only. Note: Input is given in the form of adjacency list. Example:

****

Output: [4, 5, 6, 0, 1, 2, 3]

**2. Approach:**

Solution: For a given graph, there can be many topological ordering of the vertices possible.

WHAT: Let us first see WHAT we will have to do to find topological ordering.We will perform a DFS traversal from each unvisited vertex, and add the nodes into a stack data structure in the postorder (while returning from the recursive call). All the vertices will be pushed into the stack in one of the valid topological ordering.

By adding a given element x to the stack in postorder, we are ensuring that all the vertices which is x dependent upon, must be already present in the stack, as we have traversed all of the unvisited neighbours of x, and are now returning.

We can start the DFS traversal from any vertex, but we must do DFS traversal from each unvisited vertex. Hence, no vertex should be absent in the topological ordering

HOW: Now, let us see how we can store the topological ordering by filling the stack in the postorder of recursion calls

**3. Pseudo Code:-**

1.Create a visited boolean array of size equal to number of vertices: boolean[] visited = new boolean[vtces]; 2.Initialize an empty stack of Integers: Stack< Integer> st = new Stack<>(); This stack will store the vertices in topological ordering. 3.For all unvisited nodes in {0, 1, 2, 3, ... n-1}, call for the DFS traversal with parameters as source node, visited array and the stack object.

Modified DFS Traversal

Note: We are pushing all the unvisited vertices in their postorder (while returning from the DFS call): st.push(src);

1.Finally, after all the vertices are visited and pushed into the stack, we will pop them one by one from the stack and print the node's value in separate lines. The vertices will get popped in topological ordering only.

2.WHY: Now, let's see why we are storing the elements in stack data structure, and that too in postorder.

3.Q) Why we cannot do simple DFS traversal and print the nodes in preorder or postorder. What is the need of stack data structure?

R) Let us consider the above example and do normal DFS.

If we had simply printed the vertices in preorder instead of pushing them into a stack in postorder, then since we were calling DFS for vertex 4 after calling DFS for vertex 0, we will have printed vertices [0,1,2,3] before vertices [4,5,6]. But there is an edge 4 -> 3, which will violate the condition of valid topological ordering if we will directly print the vertices in preorder.

If we directly print the nodes in postorder without storing in stack, we will print the reverse of topological sorting, as for all the edges [u -> v], v will appear before u in direct printing.

Implementation- Note: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

Edge(int src, int nbr) {

this.src = src;

this.nbr = nbr;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int vtces = Integer.parseInt(br.readLine());

ArrayList< Edge>[] graph = new ArrayList[vtces];

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

graph[i] = new ArrayList<>();

}

int edges = Integer.parseInt(br.readLine());

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

String[] parts = br.readLine().split(" ");

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

graph[v1].add(new Edge(v1, v2));

}

boolean[] visited = new boolean[vtces];

Stack< Integer> st = new Stack<>();

for (int v = 0; v < vtces; v++) {

if (visited[v] == false) {

topological(graph, v, visited, st);

}

}

while (st.size() > 0) {

System.out.println(st.pop());

}

}

public static void topological(ArrayList< Edge>[] graph,

int src, boolean[] visited, Stack< Integer> st) {

visited[src] = true;

for (Edge e : graph[src]) {

if (!visited[e.nbr]) {

topological(graph, e.nbr, visited, st);

}

}

st.push(src);

This code is written and explained by our team in the solution video . Do check it out to understand the concept completely.

**4. Analysis:**

Time Complexity:

Well, there is nothing magical here, just a simple variation of DFS to store the elements in a stack data structure. Hence the time complexity will be O(N + E) where N = number of vertices and E = number of edges.

Space Complexity:

Since, we are storing all the vertices in a stack data structure, also we will be using a visited array, the space complexity will be O(N). We are not considering the space of O(E) of the adjacency list, as it has been given to us in the form of input.

Can you guess how we can solve the problem using BFS traversal? Let me give you a hint. Try to analyze the indegree of every vertex. What will be the effect of removing a node to the indegree of its neighbouring nodes?

Hope that you liked the article on 'Order of Compilation'. Subscribe to Pepcoding's youtube channel for more such amazing video content on Data Structures & Algorithms. You can suggest any improvements to the article on our telegram channel, or on the youtube channel's comment section.