Assignment II

COURSE CODE: CSE 214

COURSE NAME: Algorithm

<u>Submitted To</u>

Subroto Nag Pinku

Department of CSE

Daffodil International University

<u>Submitted By</u>

Name: Tanjil Rahman

ID: 191-15-12536

Section: O-14

Department of CSE

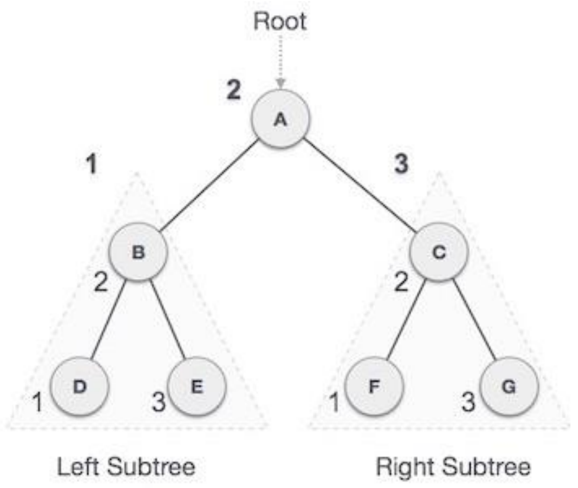
1. Full Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

1. **In-order Traversal:**
   In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself. If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

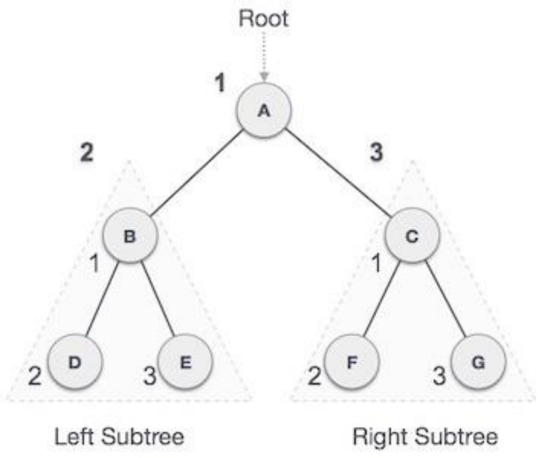


We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

$$\boldsymbol{D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G}$$

2. Pre-order Traversal:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

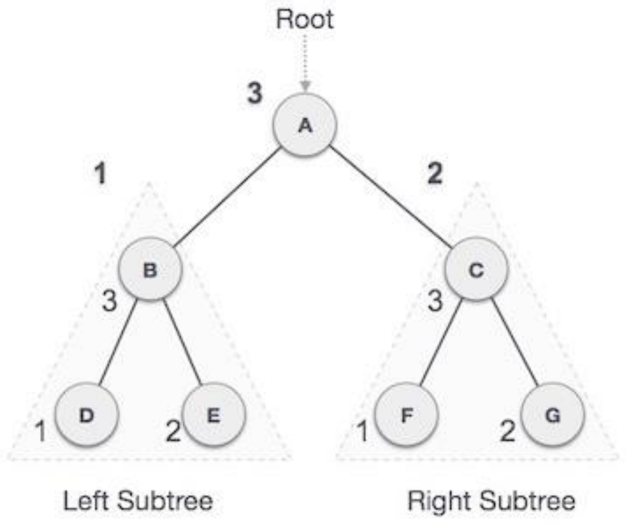


We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

3. **Post-order Traversal:**

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



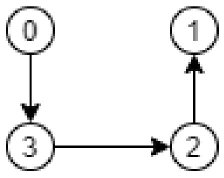
We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

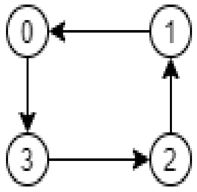
2. Cycle Finding

In graph theory, a cycle graph or circular graph is a graph that consists of a single cycle, or in other words, some number of vertices connected in a closed chain.

To determine the cycle in a graph we can implement DFS. rom every unvisited node. Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back-edge present in the graph. A back edge is an edge that is joining a node to itself (self-loop) or one of its ancestors in the tree produced by DFS.



Acyclic graph (no back edeges)



Cyclic graph (introducing back edeges)

To detect a cycle in a directed graph (i.e to find a back edge), we can use depth-first search (with some introduction of local state to tell we if a back edge occurs):

[We will maintain 3 buckets of vertices: white, grey, & black buckets. (We can also color vertices instead)]

- The white bucket will contain all of the unvisited vertices. At the start of our traversal, this means every vertex is in the white bucket.
- Before visiting a vertex, we will move it from the white bucket into the grey bucket.
- After fully visiting a vertex, it will get moved from the grey bucket into the black bucket.
- We can skip over vertices already in the black bucket, if we want to try and visit them again.
- When visiting the children/descendants of a vertex, if we come to a descendant vertex that is already in the grey bucket - that means we have found a back edge/cycle.
- This means the current vertex has a back edge to its ancestor - as we only arrived at the current vertex via its ancestor. So, we have just determined that there is more than one path between the two (a cycle).

To detect a cycle in an undirected graph, it is very similar to the approach for a directed graph. However, there are some key differences:

- We no longer color vertices/maintain buckets. Instead we use visited set to keep track the visited vertex.

- We have to make sure to account for the fact that edges are bi-directional - so an edge to an ancestor is allowed, if that ancestor is the parent vertex.

- We only keep track of visited vertices (similar to the grey bucket).

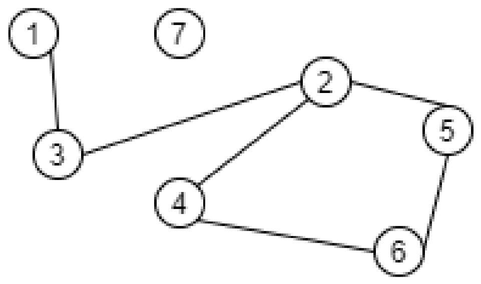
- When exploring/visiting all descendants of a vertex, if we come across a vertex that has already been visited then we have detected a cycle.

Time complexity is O (V + E) for an adjacency list. Space complexity is O(V). For an adjacency matrix, the time & space complexity would be O(V^2).

3. Component Finding

In a graph, connected components are set of vertices which are reachable. In other words, connected component is a subgraph in which any two vertices are connected by path.

Let's consider a graph shown below and find the component of the graph,



Component finding can be done by using both DFS and BFS. We will implement the DFS or BFS in a certain node and will see that if all the nodes are visited or not. If we found that there exist some nodes that are not visited at all then again, we will run BFS or DFS on that particular node.

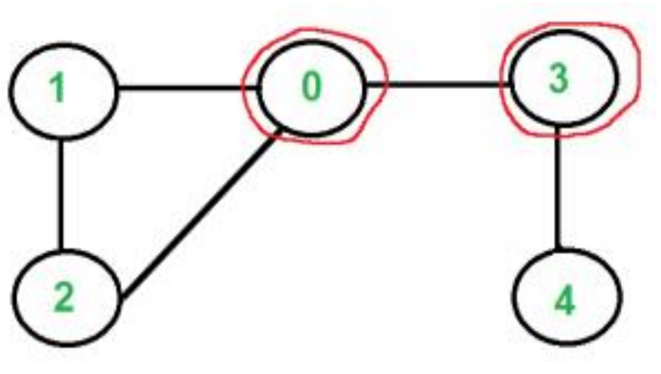
The above graph shows that If we start DFS using the vertex 1 then we can have visited all the nodes till 6. That means we found 1 connected component in the graph. But vertex 7 is not connected to any other vertex that means we have to run DFS again on this vertex and count the connected component to 1

Therefore, after summing up all the connected components in the graph we find that in the graph there are 2 connected components.

4. Articulation Point Finding

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks.
For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

Following are some example graphs with articulation points encircled with red color.

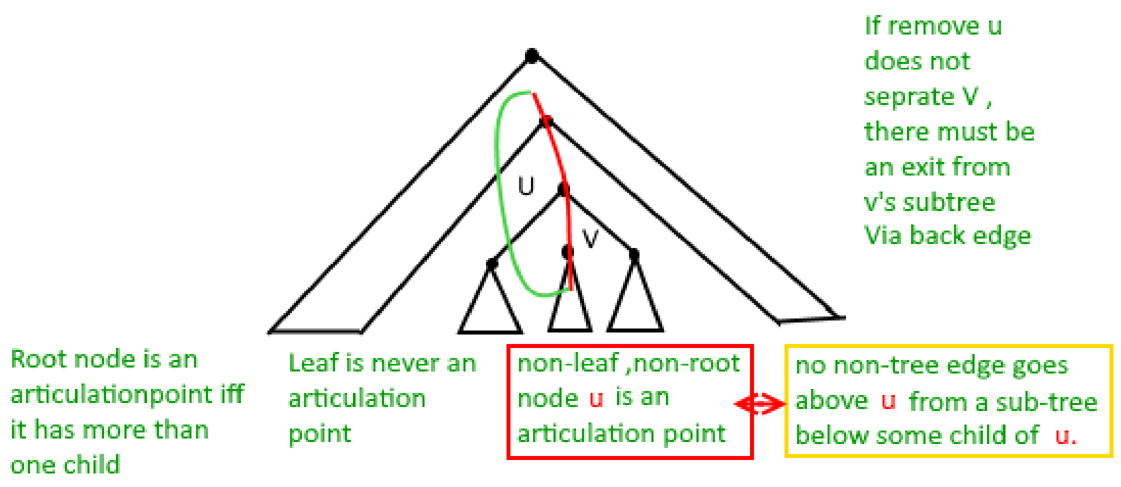


Articulation points are 0 and 3

1) For every vertex v, do following

a) Remove v from graph
b) See if the graph remains connected (We can either use BFS or DFS)
c) Add v back to the graph

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.
1) u is root of DFS tree and it has at least two children.
2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.


If remove u does not seprate V , there must be an exit from v's subtree Via back edge
U
V
Root node is an articulationpoint iff it has more than one child
Leaf is never an articulation point
non-leaf ,non-root node u is an articulation point
no non-tree edge goes above u from a sub-tree below some child of u.

Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point.

We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

5. Topological Sort

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Here we are implementing topological sort using Depth First Search.

Step 1: Create a temporary stack.

Step 2: Recursively call topological sorting for all its adjacent vertices, then push it to the stack (when all adjacent vertices are on stack). Note this step is same as Depth First Search in a recursive way.

Step 3: At last, print contents of stack.

Considering the following graph:

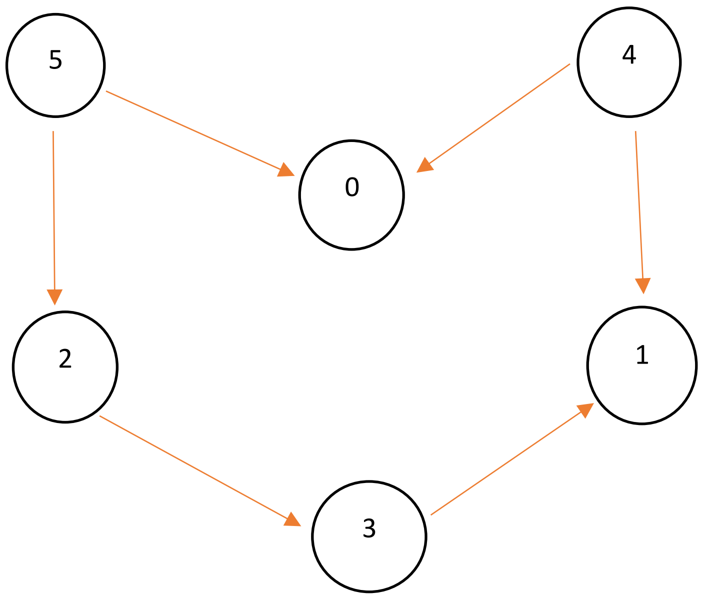
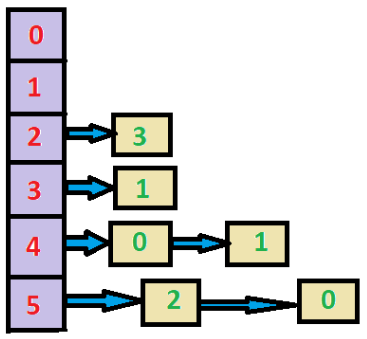


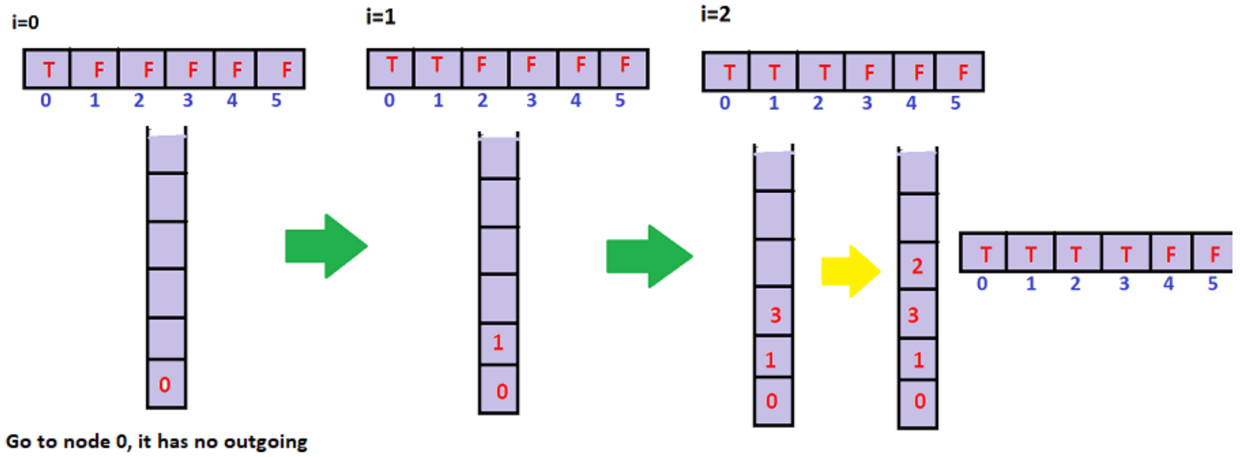
Fig 01. Graph

Here is a graph, 6 vertex and 6 edges. Now we find out the adjacency list of the graph.

Stepwise demonstration of the stack after each iteration of the loop(topologicalSort()):

Following is e adjacency list of the given graph:

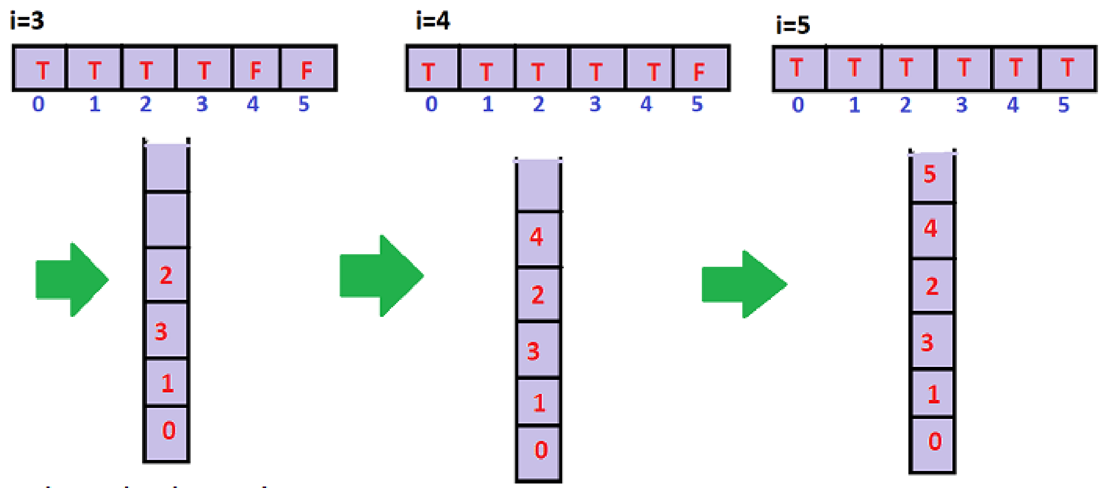




Go to node 0, it has no outgoing edges(i.e. no successors) so push node 0 into the stack and mark it visited.

Go to node 1, it has no outgoing edges(i.e. no successors) so push node 1 into the stack and mark it visited.

Go to node 2, process all the adjacent nodes(includes node) and mark node 2 visited

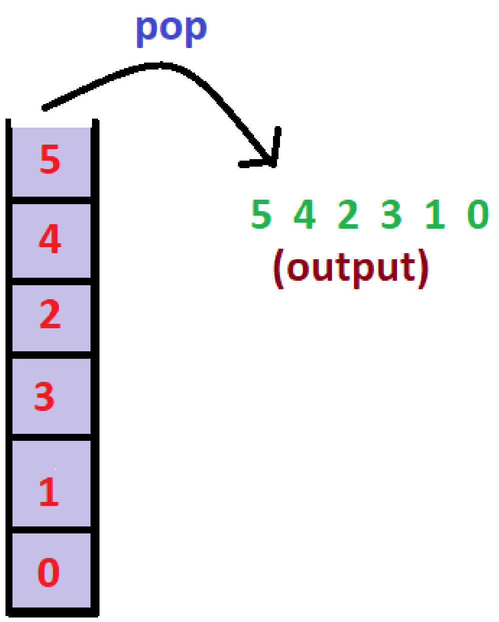


Node 3 is already visited so continue with next iteration

Go to node 4,all it's adjacent nodes are already visited so push node 4 into the stack and mark it visited.

Go to node 5, all it's adjacent nodes are already visited so push node 5 into the stack and mark it visited.

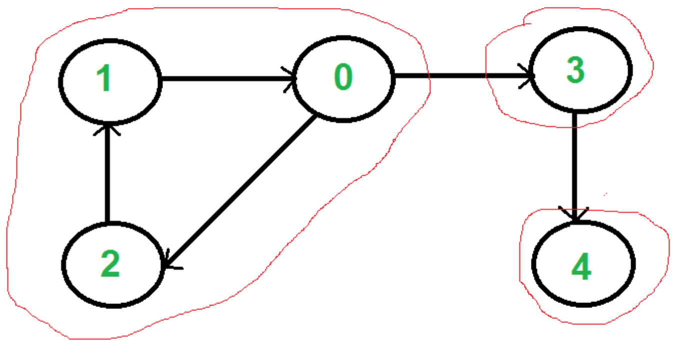
The contents of the stack are:



So, the topological sorting of the above graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the above graph is "4 5 2 3 1 0". Both of them are correct!

6. Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.

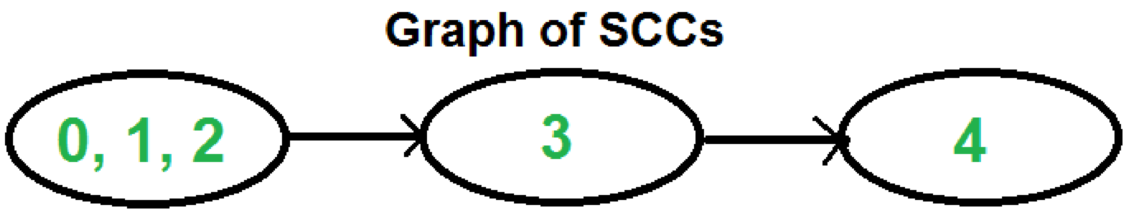
1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.

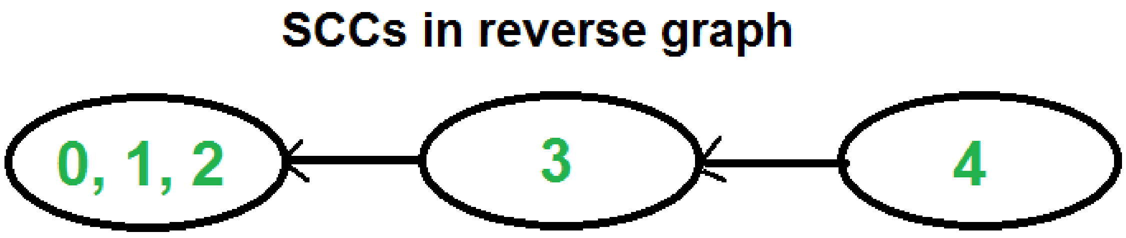
2) Reverse directions of all arcs to obtain the transpose graph.

3) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for

getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC (See this for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.
In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.


Graph of SCCs
0, 1, 2
3
4


SCCs in reverse graph
0, 1, 2
3
4