-> All below algos run in O(V+E).

**=> STRONGLY CONNECTED COMPONENTS :**

(CONCEPT FOR DIRECTED GRAPH):

**-> KOSARAJU ALGO FOR SCC:**

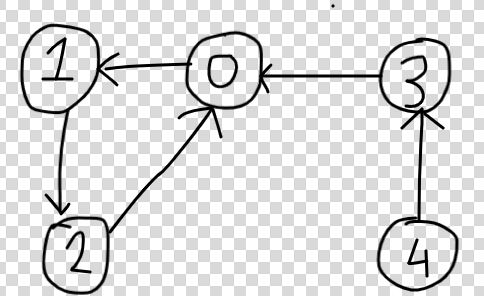
1. DO a Normal DFS on the Graph, but after calling DFS on the Adjacent Vertices of the Graph , push that Vertex on the Stack.

**The One who finishes Later will be on TOP, like topological sort.**

In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0. (0 on top)

1. Now obtain a New graph by **Reversing the Edges** of the Original Graph,

Called **TRANSPOSE GRAPH**.



1. Now we will pop elements one by one from stack and if that node is not already visited,we will run a DFS from that node.

**First of all it is ensured that at least 1 node of Every Parent(**the comp from which there is an edge to another comp**) component will be above all the nodes of the child component in the stack,and bcoz the edges are reversed now we have no chance to visit the child component from there.**

-> **So 2 facts :**

1. **Parent Component will be traversed first now(**as at least one node of it will be on top of the stack**)**
2. **No way to reach the child component now as the Edges are reversed.**

**-> Bcoz of these 2 facts, when the dfs of an Unvisited node popped from the stack finishes,we ensure that all the nodes traversed in the process will belong to 1 SCC.**

-> Now again pop another element , if it’s already visited, that means it’s component is already traversed, so move on and try the next top.

Our Stack was 1, 2, 4, 3, 0. (0 on top)

So initially, 0,1,2 will be discovered as 1 Strongly Connected Component.

Then 3 will be discovered as another.

And then 4.

Now, 1 and 2 are visited already so they are Ignored.

**-> If you treat all SCCs as nodes, after finishing, all of them will form a Directed Acyclic Graph,**bcoz if there would have been a cycle between any 1 of them, they would have been in the same strongly connected component.

**Code :** [**https://practice.geeksforgeeks.org/problems/strongly-connected-components-kosarajus-algo/1#**](https://practice.geeksforgeeks.org/problems/strongly-connected-components-kosarajus-algo/1#)

**Theoretical question for concept(not for interview) :**

[**https://www.geeksforgeeks.org/minimum-edges-required-to-make-a-directed-graph-strongly-connected/#:~:text=Explanation%3A,of%20edges%20required%20is%201.**](https://www.geeksforgeeks.org/minimum-edges-required-to-make-a-directed-graph-strongly-connected/#:~:text=Explanation%3A,of%20edges%20required%20is%201.)

**=> Questions on SCCs :**

1. [(H)Semiconnected(SCC+Condensed graph+Topo Sort)](https://docs.google.com/document/d/1Lx97AO-1YSfvAPl88JqrBVgIM6sT3mMB4J2WsULCDs0/edit)
2. [(H)Min edges needed to be added to DG to make it reachable from GIVEN vertex](https://docs.google.com/document/d/1r2zUWUB5JPetzm3ZFdamKE8bX_X_Q2Op9fsmdXiGuYk/edit)

**=> TARJAN ALGO** for SCC **:**

**PROBLEM :** [**https://cses.fi/problemset/task/1683/**](https://cses.fi/problemset/task/1683/)

**APPROACH :**

**->** Keep visiting a Path and incrementing the timer and assigning the disc and low values equal to the current timer.

-> Need a stack to maintain the nodes in the current SCC.

-> And also a boolean vector, so if v[u]=true,means that node is in the current SCC.

-> If came across a node which is already visited ‘v’, then check whether stack[v]=true(means that node is in the same component or not).

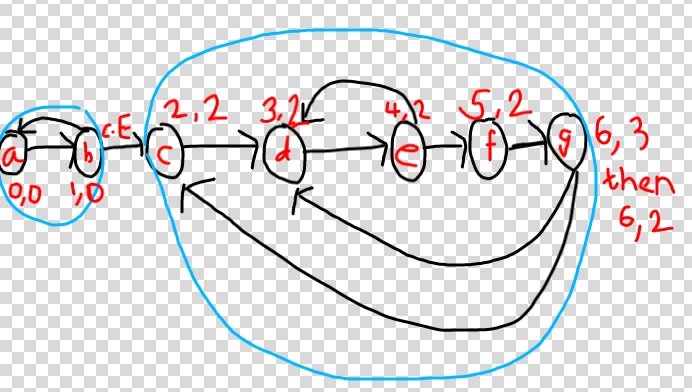
-> If YES, then update the ‘low’ of the node ‘u’ to min(low[u],disc[v]).

-> Same way , while backtracking inform the other nodes too , that we can reach a node ‘v’.

So all those nodes ‘u’ back on the path, will update their timer as min(low[u],low[v]).

-> if(low[u]==disc[u]) means we found the head of SCC,means it was the 1st node from which the dfs began , when we started processing that component. So pop all the nodes from the stack until ‘u’,so that the unidirected edges from some another component can be identified as Cross Edges.

**Dry run :**



-> Here let’s start from a.Keep updating disc and low as given.

->Assume Node g visits c first .So low = min(6,2) =2. , now it visits d, so low=min(2,3)=2.So condition low[u] = min(low[u],disc[v]) is useful,otherwise if we directly update low[u]=low[v],then low[g] would be stored as 3 as d was visited last, so while backtracking, f,e,and d would store their low as 3,and so ‘d’ will be detected as head of SCC as disc[d]=low[d],which is wrong.

-> Now while backtracking low[e] is already updated as 2,now when ‘e’ tries to visit ‘d’ , low[e]=min(2,3)=2,so here also condition low[e]=min(low[e],disc[v]) helps.

-> So while backtracking the condition : **low[u] = min(low[u],low[v])**

-> And back edge case(means finding that the node which we are going is already visited):

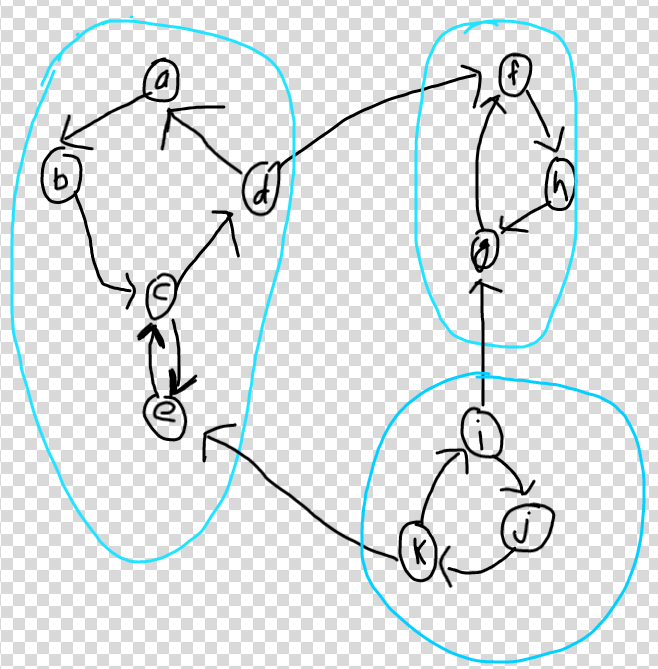
Condition -> **low[u]=min(low[u],disc[v])**

**->Now ‘c’ is detected as Head of SCC, as low[c] == disc[c]**

**-> Like we used for finding bridges, the condition if(low[v] > disc[u]) , can’t be used to discover the Head of SCC.**Bcoz let’s imagine we started from node ‘c’ in above figure, then you can’t discover head of SCC by this condition,only if we start from node 0 we can detect as the then we include the cross Edge in our path.

**CODE :** [**https://cses.fi/paste/35d70a729f0c5daf2757ff/**](https://cses.fi/paste/35d70a729f0c5daf2757ff/)

**More theory :**

****

-> The Edges joining 1 SCC to another is called Cross Edges.

**-> If we consider all SCCs as nodes then they will always form DIRECTED ACYCLIC GRAPH.**

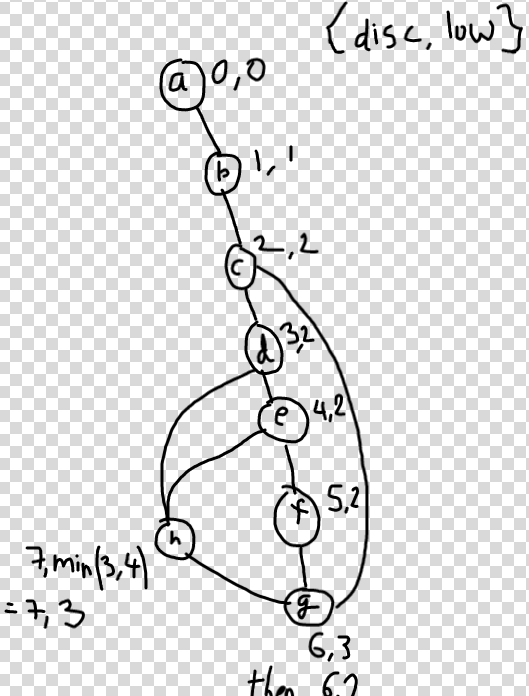
**Bridges(Cut Edge) (Concept for Undirected Graph) :**

-> A Bridge is an Edge , removing which the Undirected Graph will be divided into components.

-> If there is no BACK EDGE from a node to it’s ancestor or parent,then the Edge will be a Bridge.

**Problem :** [**https://leetcode.com/problems/critical-connections-in-a-network/**](https://leetcode.com/problems/critical-connections-in-a-network/)

**Approach :**



->’**disc’ is the discovery time at which the node was first discovered.And ‘low’ is the timer value of the minimal timer ancestor node it can reach (if any).**

-> We keep on going through a path,until we find a node which is already visited let’s say node ‘v’. So this means that node’s timer will be less than current,so we detect a Back edge.

-> In the figure ,the node ‘h’ first visits a already visited node ‘d’,,and so it updates the low value to 3, but then it visits ‘e’,so low[h]=min(3,4)=3, so here the condition low[u] = min(low[u],disc[v]) helps,otherwise if we directly update low[u]=low[v],after backtracking,then low[h] will be 4,which is wrong,bcoz actually it can go to a node having timing as low as 3,and so in future the edge 3-4 might be detected as bridge just bcoz execution assumes we cant reach 3 using back edge.

-> Now it backtracks and tells all the nodes that we can have a back edge, until the node ‘v’, by updating their low values.

-> So ‘g’ updates it’s low value to 3 , by condition low[u] = min(low[u],low[v]),but then it goes for a visit to ‘c’,and so now it has to update it’s low value to ‘2’,again by the condition low[u] = min(low[u],disc[v]),

-> And so while backtracking, until ‘d’, all nodes update their low values to ‘2’,so when backtracking reaches ‘c’, ‘c’ finds the low time of ‘d’ as 2 which is equal to it’s discovery time,and so it knows that all the nodes below it can reach BACK upto it.

-> **So basically we just keep on updating what is the minimum timer node that can be reached in the current component.**

->Now low[c] > disc[b] , so ‘b’ understands that neither ‘c’ ,nor any other nodes from ‘c’ have a Back Edge to ‘b’ or any node above ‘b’,so ‘b-c’ is a Cut Edge ,and similarly ‘a-b’

**-> If while backtracking the low[child] > disc[parent],this means that the child was not able to reach the parent or any ancestor above,so it can’t come back above through a Back edge and so that edge is a Bridge.**

-> Ignore the child-parent edge,as otherwise it will be considered as a back edge although it’s a forward edge only.

**Code :**[**https://leetcode.com/submissions/detail/528970321/**](https://leetcode.com/submissions/detail/528970321/)