Syllabus:

Graphs Algorithms: Connected Components in a graph, Finding Bridges in a Graph and Finding Articulation Point in a Graph, Maximum Flow Algorithms, Lowest Common Ancestor.

Topological Sort: Introduction, Applications: Parallel Courses, Course Schedule.

Graphs Algorithms:

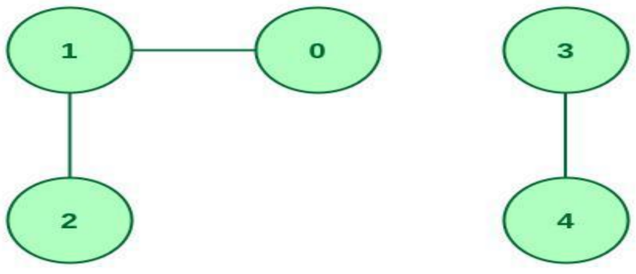
Applications:

1. **Connected Components in a graph.**
2. **Finding Bridges in a Graph.**
3. **Finding Articulation Point in a Graph.**
4. **Maximum Flow Algorithms.**
5. **Lowest Common Ancestor**.

1. Connected Components in a graph

- ➢ A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.
- ➢ Let's try to simplify it further, though. A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. The main point here is reachability.
- ➢ In connected components, all the nodes are always reachable from each other.
- ➢ Given an undirected graph, it's important to find out the number of connected components to analyze the structure of the graph – it has many real-life applications. We can use either DFS or BFS for this task.

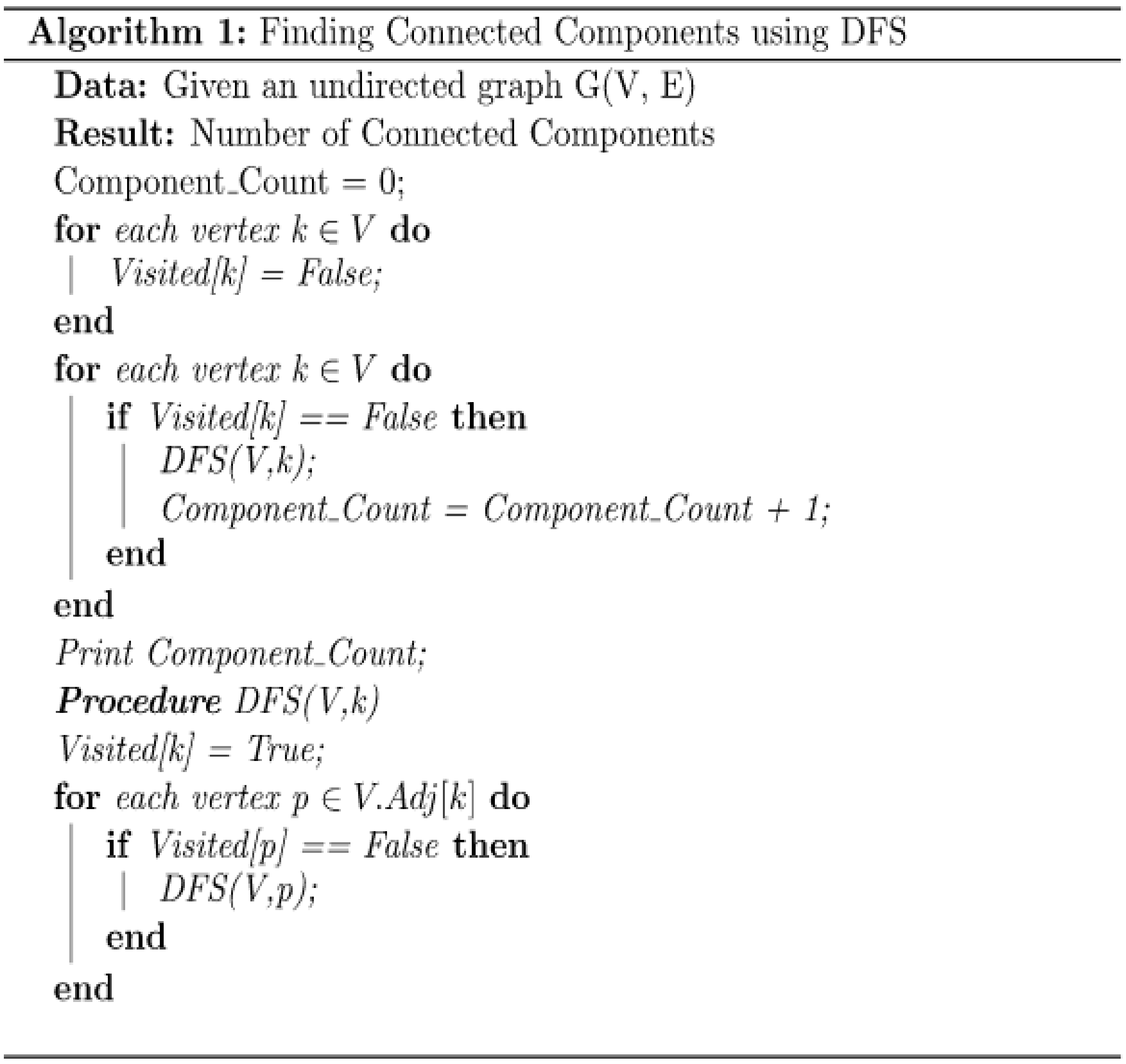
Example: The undirected graph has two connected components shown below



Input : 5 3 // number of cities(5){(0,1,2,3,4)**} and routes(3)**
0 1 //city1, city2 connections
1 2
3 4
Output : 2

Explanation: There are 2 different connected components. They are {0, 1, 2} and {3, 4}.

- **In this section, we'll discuss a DFS-based algorithm that gives us the number of connected components for a given undirected graph:**



```
Algorithm 1: Finding Connected Components using DFS
Data: Given an undirected graph G(V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex $k \in V$ do
    Visited[k] = False;
end
for each vertex $k \in V$ do
    if Visited[k] == False then
        DFS(V,k);
        Component_Count = Component_Count + 1;
    end
end
Print Component_Count;
Procedure DFS(V,k)
Visited[k] = True;
for each vertex $p \in V.Adj[k]$ do
    if Visited[p] == False then
        DFS(V,p);
    end
end
```

Java Program for Finding Connected Components in graph using DFS:

ConnectedComponents.java

```
import java.util.*;

class ConnectedComponents

{
	// A graph is an array of adjacency lists.
	// Size of array will be V (number of vertices in graph)
	int V;
	ArrayList<ArrayList<Integer> > adjListArray;

	// constructor
	ConnectedComponents(int V)
	{
		this.V = V;
		// define the size of array as     number of vertices

		adjListArray = new ArrayList<>();
		// Create a new list for each vertex such that adjacent nodes can be stored
		for (int i = 0; i < V; i++)
	    {
			adjListArray.add(i, new ArrayList<>());
		}
	}

	// Adds an edge to an undirected graph
	void addEdge(int src, int dest)
	{
		// Add an edge from src to dest.
		adjListArray.get(src).add(dest);

		// Since graph is undirected, add an edge from dest to src also
		adjListArray.get(dest).add(src);
	}

	void DFSUtil(int v, boolean[] visited)
	{
		// Mark the current node as visited and print it
		visited[v] = true;
```

```
                System.out.print(v + " ");
                // Recur for all the vertices
                // adjacent to this vertex
                for (int x : adjListArray.get(v))
                    {
                        if (!visited[x])
                                DFSUtil(x, visited);
                }
        }
        void connectedComponents()
        {
                // Mark all the vertices as not visited
                boolean[] visited = new boolean[V];
                for (int v = 0; v < V; ++v) {
                        if (!visited[v]) {
                                // print all reachable vertices
                                // from v
                                DFSUtil(v, visited);
                                System.out.println();
                        }
                }
        }
public static void main(String args[])
   {
     Scanner sc=new Scanner(System.in);
     System.out.println("enter number of vertices ");
     int v=sc.nextInt();
        System.out.println("enter number of  edges");
     int e=sc.nextInt();
     ConnectedComponents g = new ConnectedComponentsh(v);
     System.out.println("enter edges");
     for(int i=0;i<e;i++)
     {
        int end1=sc.nextInt();
        int end2=sc.nextInt();
        g.addEdge(end1,end2);
     }
                System.out.println("Following are connected components");
                g.connectedComponents();
        }
```

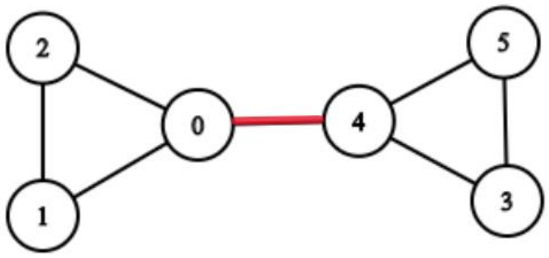
```
}
```

Sample input & output:

```
enter number of vertices
5
enter number of  edges
3
enter edges
1
0
2
1
3
4
Following are connected components
0 1 2
3 4
```

2. Finding Bridges in a Graph:

- Given an undirected graph of V vertices and E edges. Our task is to find all the bridges in the given undirected graph.
- A bridge in any graph is defined as an edge which, when removed, makes the graph disconnected (or more precisely, increases the number of connected components in the graph).
- The algorithm described here is based on DFS and has O(N+M) complexity, where N is the number of vertices and M is the number of edges in the graph.
- For Example: If the given graph is :



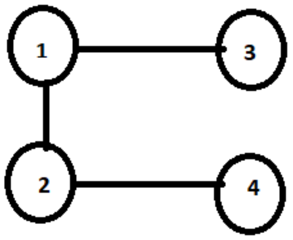
- The edge between 0 and 4 is the bridge because if the edge between 0 and 4 is removed, then there will be no path left to reach from 0 to 4.and makes the graph disconnected, and increases the number of connected components.
- **Note :**
  - There are no self-loops(an edge connecting the vertex to itself) in the given graph.

  - There are no parallel edges i.e no two vertices are directly connected by more than 1 edge.

Sample Input-1:
----
4 3 //No. of nodes =4 {1,2,3,4} & No of Edges=3
1 2
1 3
2 4
1 //P value

Sample Output-1:

2



Explanation:

There is only one path 1->2->4. so answer = 2

The algorithm steps are as follows:

1. First, we need to create the adjacency list for the given graph from the edge information(If not already given). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node.

2. Then we will start DFS from node 0(assuming the graph contains a single component otherwise, we will call DFS for every component) with parent -1.

   2.1. Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. The timer may be initialized to 0 or 1.
   2.2. Now, it's time to visit the adjacent nodes.
      2.2.1. If the adjacent node is the parent itself, we will just continue to the next node.
      2.2.2.If the adjacent node is not visited, we will call DFS for the adjacent node with the current node as the parent.
      After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.
      Now, we will check if the lowest time of insertion of the adjacent node is greater than the time of insertion of the current node.
      If it is, then we will store the adjacent node and the current node in our answer array as they are representing the bridge.
      2.2.3.If the adjacent node is already visited, we will just compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.

3. Finally, our answer array will store all the bridges.

Note: We are not considering the parent's insertion time during calculating the lowest insertion time as we want to check if any other path from the node to the parent exists excluding the edge we intend to remove.

Java Program for Finding Bridges in undirected graph using DFS:

FindingBridges.java

```
import java.io.*;
import java.util.*;
import java.util.LinkedList;

class FindingBridges
{
   private int V;   // No. of vertices

   // Array  of lists for Adjacency List Representation

   private LinkedList<Integer> adj[];
   int time = 0;
```

```
    static final int NIL = -1;

    // Constructor    @SuppressWarnings("unchecked")

FindingBridges (int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }


    // Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
        adj[w].add(v);    //Add v to w's list
    }

    // A recursive function that finds and prints bridges     // using DFS traversal
    // u --> The vertex to be visited next
    // visited[] --> keeps track of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree

    void bridgeUtil(int u, boolean visited[], int disc[], int low[], int parent[])
    {
        // Mark the current node as visited
        visited[u] = true;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;

        // Go through all vertices adjacent to this
        Iterator<Integer> i = adj[u].iterator();
        while (i.hasNext())
        {
            int v = i.next();  // v is current adjacent of u
            // If v is not visited yet, then make it a child of u in DFS tree and recur for it.
            // If v is not visited yet, then recur for it

            if (!visited[v])
            {
                parent[v] = u;
                bridgeUtil(v, visited, disc, low, parent);

                // Check if the subtree rooted with v has a connection to one of the ancestors of u

                low[u]  = Math.min(low[u], low[v]);

                // If the lowest vertex reachable from subtree
            // under v is below u in DFS tree, then u-v is  a bridge

                if (low[v] > disc[u])
                    System.out.println(u+" "+v);
            }
            // Update low value of u for parent function calls.
                    else if (v != parent[u])
                low[u]  = Math.min(low[u], disc[v]);
        }
    }
    // DFS based function to find all bridges. It uses recursive function bridgeUtil()

    void bridge()
    {
        // Mark all the vertices as not visited
        boolean visited[] = new boolean[V];
        int disc[] = new int[V];
        int low[] = new int[V];
        int parent[] = new int[V];

        // Initialize parent and visited, and ap(articulation point) arrays

        for (int i = 0; i < V; i++)
        {
            parent[i] = NIL;
            visited[i] = false;
        }
```

```
        // Call the recursive helper function to find Bridges  in DFS tree rooted with vertex 'i'
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                bridgeUtil(i, visited, disc, low, parent);
    }

    public static void main(String args[])
    {
         Scanner sc=new Scanner(System.in);
        System.out.println("enter number of vertices ");
        int v=sc.nextInt();
            System.out.println("enter number of  edges");
        int e=sc.nextInt();

      FindingBridges  g = new FindingBridges (v);
        System.out.println("enter edges");
        for(int i=0;i<e;i++)
        {
            int end1=sc.nextInt();
            int end2=sc.nextInt();
            g.addEdge(end1,end2);
        }
          System.out.println("Bridges in graph");
        g.bridge();

    }
}
```

Sample input & output:

```
enter number of vertices
5
enter number of  edges
5
enter edges
1 0
0 2
2 1
0 3
3 4
Bridges in graph
3 4
0 3
```

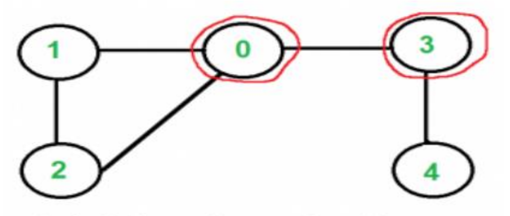
3. Finding Articulation Point in a Graph:

- A vertex is said to be an articulation point in a graph if removal of the vertex and associated edges disconnects the graph. So, the removal of articulation points increases the number of connected components in a graph.

- Articulation points are sometimes called cut vertices. The main aim here is to find out all the articulations points in a graph.

- **Example:** Given an undirected graph G, then find all the articulation points in the graph.

  Input: { Below Graph}



Articulation points are 0 and 3

Output = 0,3

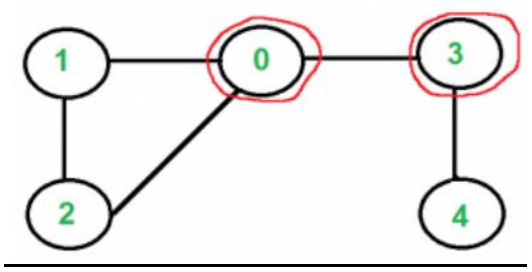
Algorithm:

Algorithm 1: Algorithm to Find all the Articulation Points

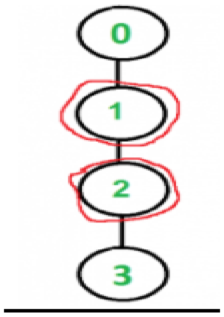
```
Data: $G(V, E), s$
Result: Articulation vertices in G
DFSTree(G);
Procedure ArticulationPoints(s, d)
Visited[s] = TRUE;
depth[s] = d;
low[s] = d;
for each $k \in Adj(s)$ do
    if Visited[s] == FALSE then
        ArticulationPoint(k, d+1);
    end
    low[s] = min (low[s], low[k]);
    if $low[k] \geq depth[s]$ then
        if $s$ = not a root node OR number of children(s) > 1 then
            print("s is a articulation point");
        end
    end
end
```

- This is a **DFS** based algorithm to find all the articulation points in a graph. Given a graph, **the algorithm first constructs a DFS tree.**
- Initially, the algorithm chooses any random vertex to start the algorithm and marks its status as visited. **The next step is to calculate the depth of the selected vertex.** The depth of each vertex is the order in which they are visited.
- **Next, we need to calculate the lowest discovery number.** This is equal to the depth of the vertex reachable from any vertex by considering one back edge in the DFS tree. An edge is a back edge if is an ancestor of edge but not part of the DFS tree. But the edge is a part of the original graph.
- **After calculating the depth and lowest discovery number for the first picked vertex, the algorithm then searches for its adjacent vertices.** It checks whether the adjacent vertices are already visited or not. If not, then the algorithm marks it as the current vertex and calculates its depth and lowest discovery number.

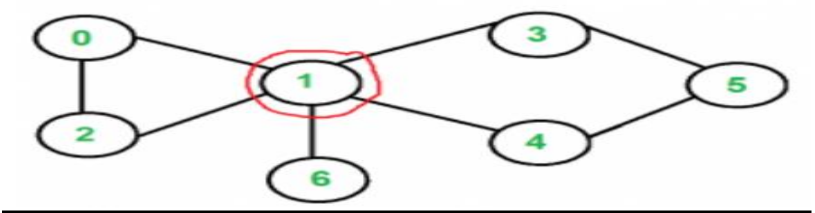
Graph 1: **(Articulation Point Graph 1: 0,3)**



Graph 2: **(Articulation Point Graph 2: 1,2)**



Graph 3: **(Articulation Point Graph 3: 1)**



Java Program for Finding Articulation Points : **ArticulationPoint.java**

```
import java.util.*;

class ArticulationPoint
 {
        static int time;

        static void addEdge(ArrayList<ArrayList<Integer> > adj, int u, int v)
        {
                adj.get(u).add(v);
                adj.get(v).add(u);
        }

static void APUtil(ArrayList<ArrayList<Integer> > adj, int u,boolean visited[], int disc[], int
low[],int parent, boolean isAP[])
        {
                // Count of children in DFS Tree
                int children = 0;

                // Mark the current node as visited
                visited[u] = true;

                // Initialize discovery time and low value
                disc[u] = low[u] = ++time;

                // Go through all vertices adjacent to this
                for (Integer v : adj.get(u))
                {
                // If v is not visited yet, then make it a child of u in DFS tree and recur for it
                        if (!visited[v])
                        {
                                children++;
                                APUtil(adj, v, visited, disc, low, u, isAP);

                // Check if the subtree rooted with v has a connection to one of the ancestors of u
                                low[u] = Math.min(low[u], low[v]);

               // If u is not root and low value of one of its child is more than discovery value of u.
                                if (parent != -1 && low[v] >= disc[u])
                                        isAP[u] = true;
                        }

                        // Update low value of u for parent function calls.
                        else if (v != parent)
                                low[u] = Math.min(low[u], disc[v]);
                }

                // If u is root of DFS tree and has two or more children.
                if (parent == -1 && children > 1)
                        isAP[u] = true;
        }

        static void AP(ArrayList<ArrayList<Integer> > adj, int V)
        {
                boolean[] visited = new boolean[V];
                int[] disc = new int[V];
                int[] low = new int[V];
                boolean[] isAP = new boolean[V];
                int time = 0, par = -1;

                // Adding this loop so that the code works even if we are given disconnected graph
                for (int u = 0; u < V; u++)
                        if (visited[u] == false)
                                APUtil(adj, u, visited, disc, low, par, isAP);

                for (int u = 0; u < V; u++)
                        if (isAP[u] == true)
                                System.out.print(u + " ");
                System.out.println();
        }

        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);
                 System.out.println("enter number of vertices ");
                int V=sc.nextInt();
                System.out.println("enter number of  edges");
                int e=sc.nextInt();
                 ArticulationPoint g = ArticulationPoint ();
                ArrayList<ArrayList<Integer> > adj1 = new ArrayList<ArrayList<Integer> >(V);

                for (int i = 0; i < V; i++)
                   {
                        adj1.add(new ArrayList<Integer>());
                  }
          System.out.println("enter edges");
         for(int i=0;i<e;i++)
         {
                 int end1=sc.nextInt();
                  int end2=sc.nextInt();
```

```
                         g.addEdge(adj1,end1,end2);
                }

                        System.out.println("Articulation points in first graph");
                        AP(adj1, V);
            }
}
```

```
enter number of vertices
5
enter number of  edges
5
enter edges
1
0
0
2
2
1
0
3
3
4
Articulation points in first graph
0 3
```

4. Maximum Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- The max flow problem is a classic optimization problem in graph theory that involves finding the maximum amount of flow that can be sent through a network of pipes, channels, or other pathways, subject to capacity constraints.
- The problem can be used to model a wide variety of real-world situations, such as transportation systems, communication networks, and resource allocation.
- In the max flow problem, we have a directed graph with a source node s and a sink node t, and each edge has a capacity that represents the maximum amount of flow that can be sent through it. The goal is to find the maximum amount of flow that can be sent from s to t, while respecting the capacity constraints on the edges.

One common approach to solving the max flow problem is the **Ford-Fulkerson algorithm**, which is based on the idea of augmenting paths. The algorithm starts with an initial flow of zero, and iteratively finds a path from s to t that has available capacity, and then increases the flow along that path by the maximum amount possible. This process continues until no more augmenting paths can be found.

Another popular algorithm for solving the max flow problem is the **Edmonds-Karp algorithm**, which is a variant of the Ford-Fulkerson algorithm that uses breadth-first search to find augmenting paths, and thus can be more efficient in some cases.

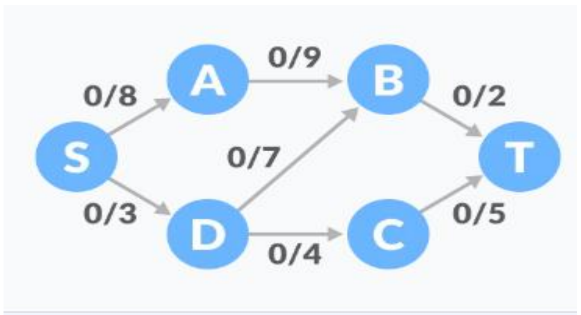
Ford-Fulkerson method:

The algorithm follows:

1. Initialize the flow in all the edges to 0.
2. While there is an augmenting path between the source and the sink, add this path to the flow.
3. Update the residual graph.

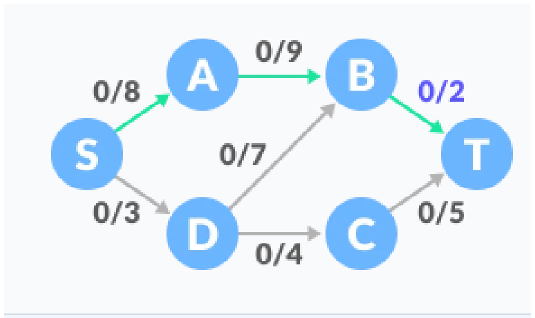
Ford-Fulkerson Example:

Step 0: The flow of all the edges is 0 at the beginning.



Flow network graph example

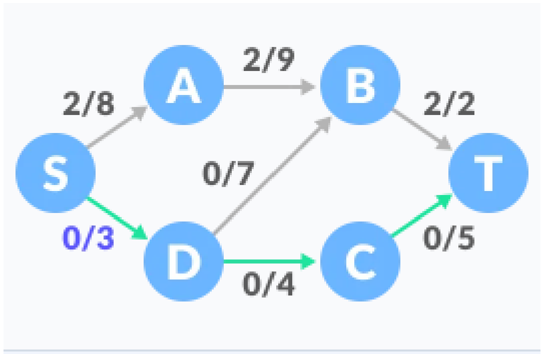
Step 1: Select any arbitrary path from S to T. In this step,we have selected path **S-A-B-T**



Find a path

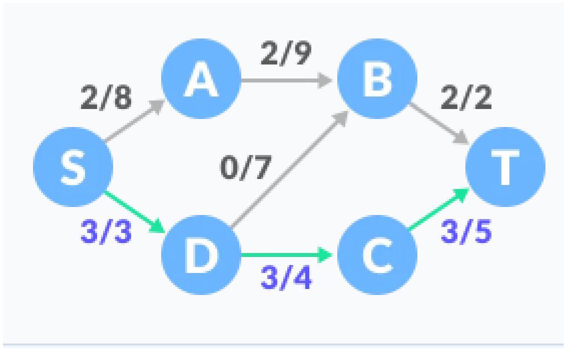
The minimum capacity among the three edges is 2 (B-T) Based on this, update the flow/capacity for each path.

Step2: Select another path **S-D-C-T**. The minimum capacity among these edges is 3**(S-D)**



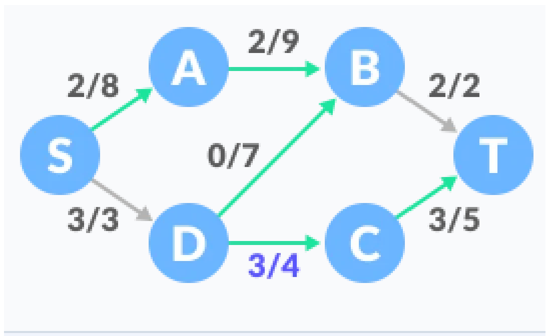
Find next path

Update the capacities according to this.



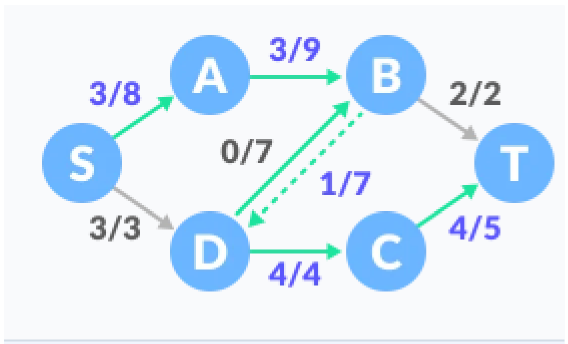
Update the capacities

Step 3: Now, let us consider the reverse-path B-D as well. Selecting path **S-A-B-D-C-T.** The minimum residual capacity among the edges is 1 **(D-C).**



Find next path

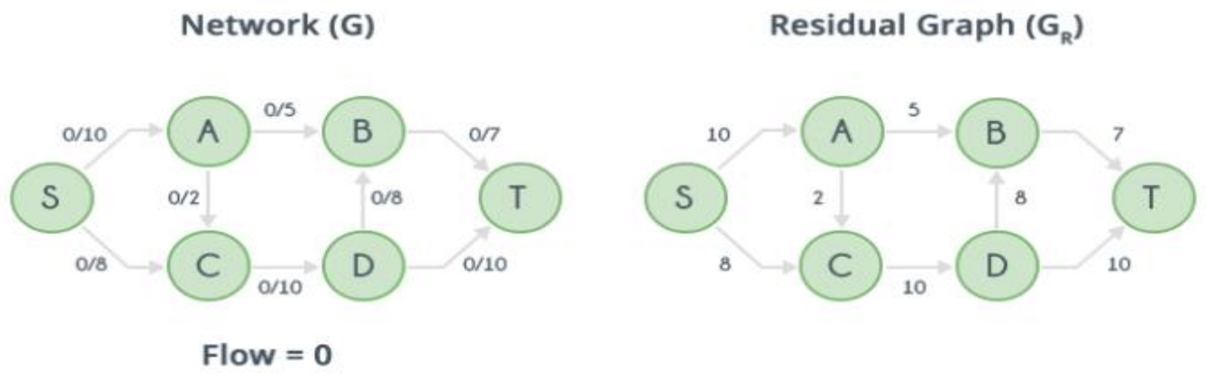
Updating the capacities.



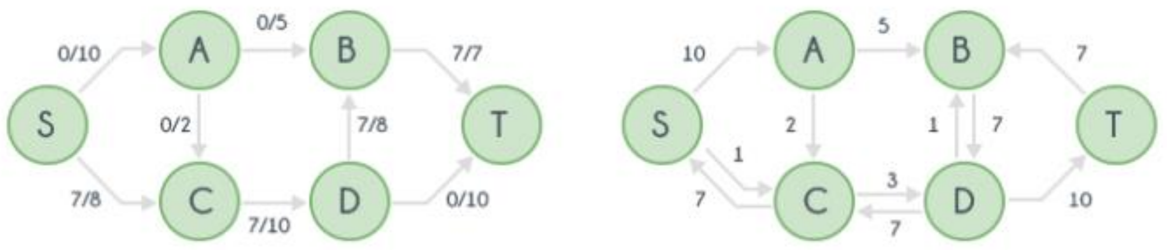
Update the capacities

Step 4: Adding all the flows = 2 + 3 + 1 = 6, which is the maximum possible flow on the flow network.

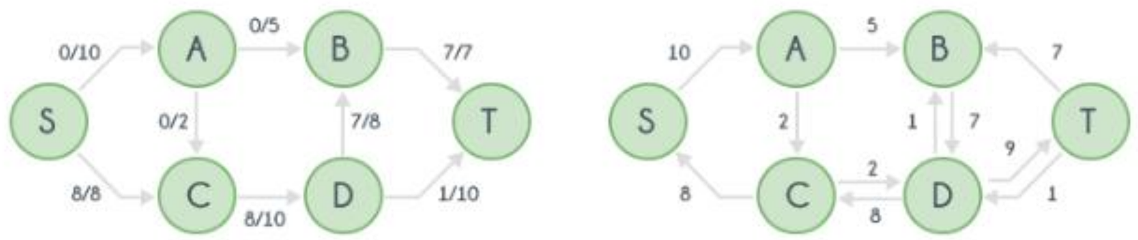
Example 2:



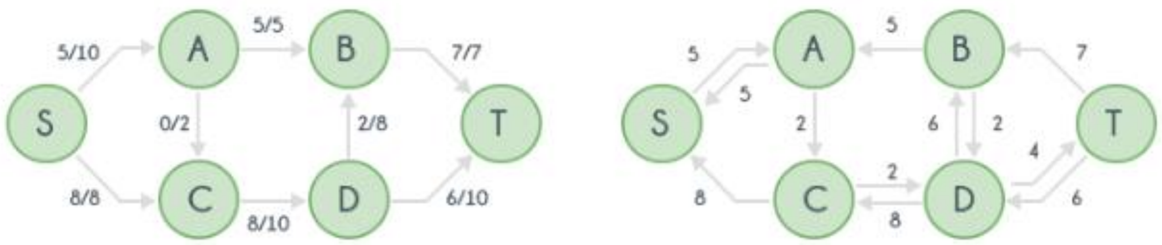
Path 1: S - C - D - B - T $\longrightarrow$ Flow = Flow + 7

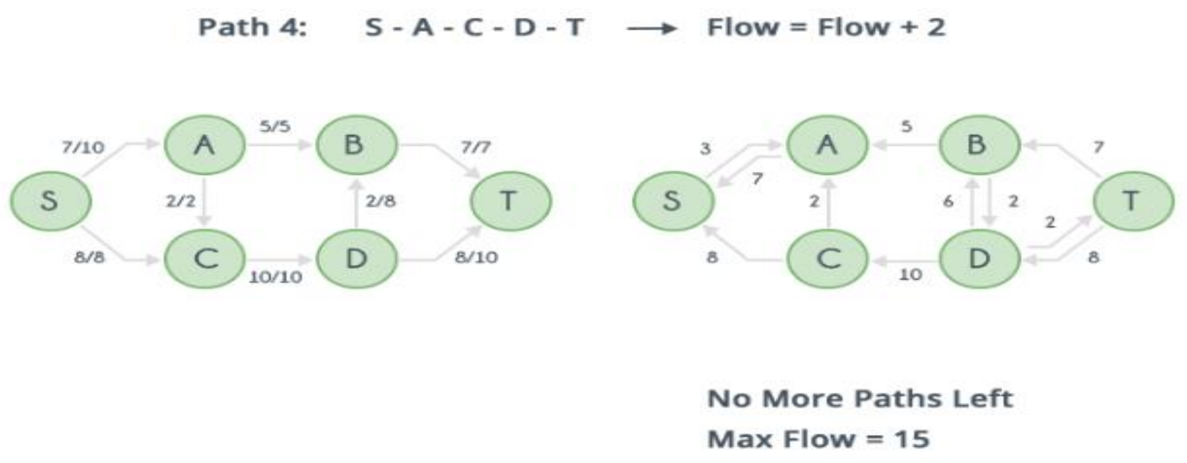


Path 2: S - C - D - T $\longrightarrow$ Flow = Flow + 1



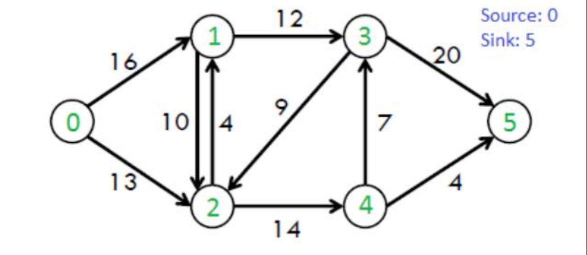
Path 3: S - A - B - T $\longrightarrow$ Flow = Flow + 5





<u>Java program for implementation of Ford Fulkerson algorithm</u>: MaxFlow.java

<u>Input:</u> 6

0 16 13 0 0 0
0 0 10 12 0 0
0 4 0 0 14 0
0 0 9 0 0 20
0 0 0 7 0 4
0 0 0 0 0 0
0 5
	 Source: 0
	Sink: 5
	16
	12
	20
	10
	4
	9
	7
	13
	14
	4

<u>Output:</u> 23

```
import java.util.*;
public class MaxFlow
{
   static int V; // number of vertices in the graph

// method to find the maximum flow in a flow network using the Edmonds-Karp algorithm
   static int findMaxFlow(int[][] graph, int source, int sink)
   {
      int[][] residualGraph = new int[V][V];
      for (int i = 0; i < V; i++)
                  {
         for (int j = 0; j < V; j++)
```

```
                {
            residualGraph[i][j] = graph[i][j];
          }
        }
         int[] parent = new int[V];
        int maxFlow = 0;

        while (bfs(residualGraph, source, sink, parent))
             {
          int pathFlow = Integer.MAX_VALUE;
          for (int v = sink; v != source; v = parent[v])
                {
            int u = parent[v];
            pathFlow = Math.min(pathFlow, residualGraph[u][v]);
          }
               for (int v = sink; v != source; v = parent[v])
                {
            int u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
          }
               maxFlow += pathFlow;
        }
            return maxFlow;
      }
        // helper method to find the shortest augmenting path in the residual graph using BFS
static boolean bfs(int[][] residualGraph, int source, int sink, int[] parent)
{
        boolean[] visited = new boolean[V];
        Arrays.fill(visited, false);
        Queue<Integer> queue = new LinkedList<Integer>();
        queue.add(source);
        visited[source] = true;
        parent[source] = -1;

        while (!queue.isEmpty())
               {
          int u = queue.poll();
          for (int v = 0; v < V; v++)
                {
```

```
            if (!visited[v] && residualGraph[u][v] > 0)
                        {
              queue.add(v);
              parent[v] = u;
              visited[v] = true;
            }
          }
        }
        return visited[sink];
    }
    public static void main(String[] args)
    {
          Scanner s=new Scanner(System.in);
      System.out.println("Enter number of vertices");
          V=s.nextInt();
      int[][] graph = new int[V][V];
          System.out.println("Enter the adjacency matrix of the directed graph");
          for(int i=0;i<V;i++)
          for(int j=0;j<V;j++)
          graph[i][j]=s.nextInt();
          System.out.println("Enter source and sink");
      int source = s.nextInt();
          int sink = s.nextInt();
      int maxFlow = findMaxFlow(graph, source, sink);
      System.out.println(maxFlow);
    }
}
```

Input =

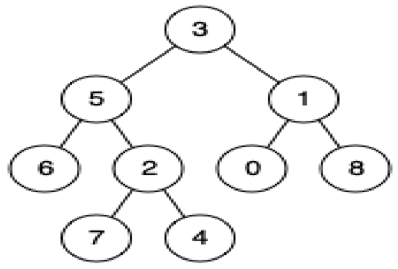
```
Enter number of vertices
6
Enter the adjacency matrix of the directed graph
0 16 13  0   0   0
0  0  10 12  0   0
0  4   0   0  14  0
0  0   9   0   0  20
0  0   0   7   0  4
0  0   0   0   0  0
Enter source and sink
0 5
```

Output = 23

➢ Lowest Common Ancestor (Binary Tree):

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes 'p' and 'q' as the lowest node in 'T' that has both 'p' and 'q' as descendants (where we allow a node to be a descendant of itself)."

Example-1:-

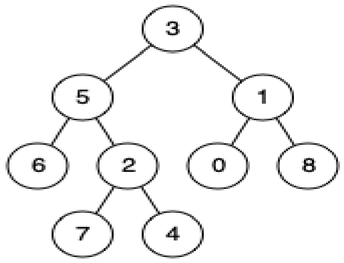


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3, since a node can be a descendant of itself according to the LCA definition.

Example-2:-



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Example 3:
Input: root = [1,2], p = 1, q = 2
Output: 1

Approach:

- If root is null or if root is x or if root is y then return root
- Made a recursion call for both

i) Left subtree

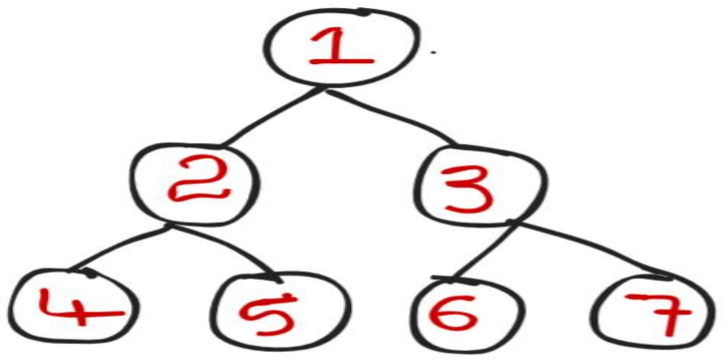
ii)Right subtree

Because we would find LCA in the left or right subtree only.

- If the left subtree recursive call gives a null value that means we haven't found LCA in the left subtree, which means we found LCA on the right subtree. So we will return right.
- If the right subtree recursive call gives null value, that means we haven't found LCA on the right subtree, which means we found LCA on the left subtree. So we will return left .
- If both left & right calls give values (not null) that means the root is the LCA.

Let's take an example and will try to understand the approach more clearly:

Consider the following Binary Tree



Example:
Input: x = 4, y = 5
Output: 2

LCA of (x,y) = > (4,5) = ? (from above given example)

Root is 1 which is not null and x,y is not equal to root, So the 1st statement in approach will not execute.

i) Call left subtree, While calling recursively it will find 4 and this call will return 4 to its parent

Point to Note: At present, the root is 2 (Look at below recursion tree for better understanding)

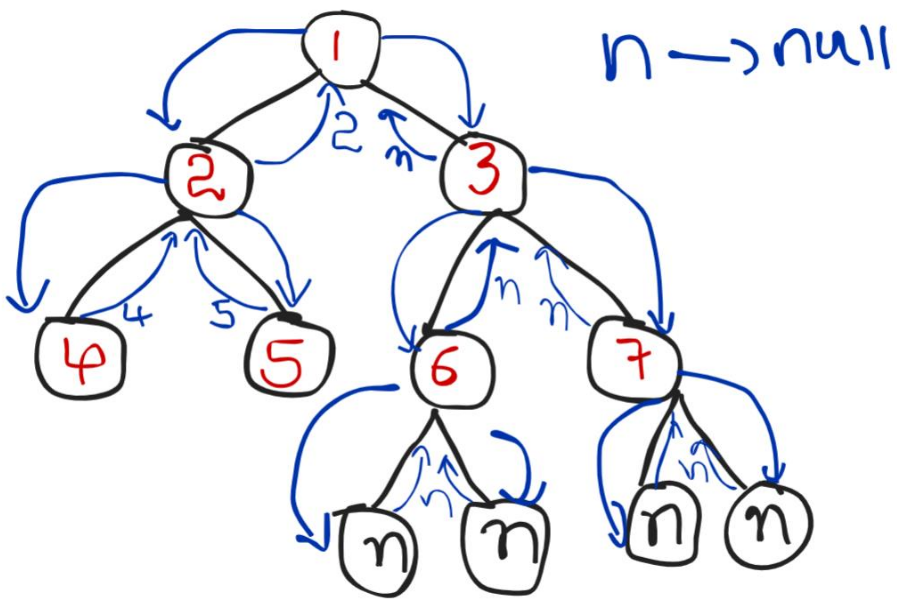
i) Call the right subtree (i.e right of 2), While calling recursively it will find 5 and this call will return 5 to its parent.

- Now the left recursive call returns value (not null) i.e 4 and also the right recursive call returns value (not null) i.e 5 to its root (at present root is 2) , and this 2 will return itself to its root i.e to 1 (main root).

Point to Note: At present, the root is 1 (Look at below recursion tree for better understanding)

- Now, the left subtree gives a value i.e 2.
- Right recursive call will give null value .because x,y are not present in the right subtree.
- As we know if the right recursive call gives null then we return the answer which we got from the left call, So we will return 2.
- Hence LCA of (4,5) is 2.

For a better understanding of the above example (LCA OF 4,5) :



Time complexity: O(N) where n is the number of nodes.

Space complexity: O(N), auxiliary space.

Java Program for Implementing Lowest Common Ancestor:

LowestCommonAncestor.java

```
import java.util.*;
class BinaryTreeNode
{
            public int data;
            public BinaryTreeNode left, right;
            public BinaryTreeNode(int data)
           {
                        this.data = data;
                        left = null;
                        right = null;
            }
}

class LowestCommonAncestor
{
            static int depth[],parent[];
            static boolean visited[];
            static BinaryTreeNode root;
            //static BinaryTreeNode temp = root;
    static void insert(BinaryTreeNode temp, int key)
    {
        if (temp == null)
        {
            root = new BinaryTreeNode(key);
            return;
        }
        Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
        q.add(temp);

        // Do level order traversal until we find an empty place.
        while (!q.isEmpty())
        {
            temp = q.peek();
            q.remove();
             if (temp.left == null)
            {
                temp.left = new BinaryTreeNode(key);
                break;
            }
            else
                q.add(temp.left);

            if (temp.right == null)
            {
                temp.right = new BinaryTreeNode(key);
                break;
            }
            else
                q.add(temp.right);
        }
}
public static void dfs(BinaryTreeNode root)
{
            visited[root.data]=true;
            if(root.left!=null)
            {
                        depth[root.left.data]=depth[root.data]+1;
                        parent[root.left.data]=root.data;
                        dfs(root.left);
            }
            if(root.right!=null)
            {
                        depth[root.right.data]=depth[root.data]+1;
                        parent[root.right.data]=root.data;
                        dfs(root.right);
            }
}

static BinaryTreeNode lca(BinaryTreeNode root,BinaryTreeNode u, BinaryTreeNode v)
{
            while(depth[u.data]!=depth[v.data])
            {
                        if(depth[u.data]>depth[v.data])
                        {
                                    depth[u.data]=depth[parent[u.data]];
                                    u.data=parent[u.data];
                        }
                        else
                {
                        depth[v.data]=depth[parent[v.data]];
                        v.data=parent[v.data];
                }
        }
        while(u.data!=v.data)
        {
                u.data=parent[u.data];
                v.data=parent[v.data];
        }
        return u;
}
public static void main(String args[])
   {
        Scanner sc=new Scanner(System.in);
        String str[]=sc.nextLine().split(" ");
        root=new BinaryTreeNode(Integer.parseInt(str[0]));
        for(int i=1 ; i<str.length; i++)
        insert(root,Integer.parseInt(str[i]));
        BinaryTreeNode p=new BinaryTreeNode(sc.nextInt());
        BinaryTreeNode q=new BinaryTreeNode(sc.nextInt());
        depth=new int[100];
        parent=new int[100];
        visited=new boolean[100];
        depth[root.data]=1;
        dfs(root);
        BinaryTreeNode res=lca(root,p,q);
        System.out.println(res.data);
}
}
```

Input=1 2 3 4 5 6 7 8 9 10 11
7 8
Output=1
Input=11 99 88 77 22 33 66 55 10 20 30 40 50 60 44
66 55
Output=11

Topological Sort:

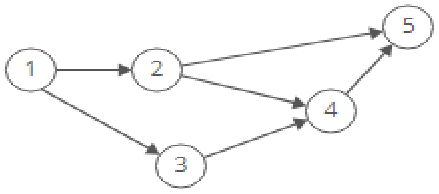
Applications:

1. **Parallel Courses.**
2. **Course Schedule.**

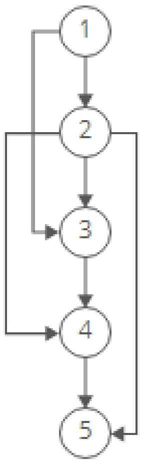
Introduction:

- Topological Sorting or Kahn's algorithm is an algorithm that orders a directed acylic graph in a way such that each node appears before all the nodes it points to in the returned order, i.e. if we have a --> b, a must appear before b in the topological order.
- It's main usage is to detect cycles in directed graphs, since no topological order is possible for a graph that contains a cycle. Some of it's uses are: deadlock detection in OS, Course schedule problem etc.
- The **topological sort** algorithm takes a directed graph and returns an array of the nodes where each node appears *before* all the nodes it points to.
- The ordering of the nodes in the array is called a topological ordering.

  Example:



- Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

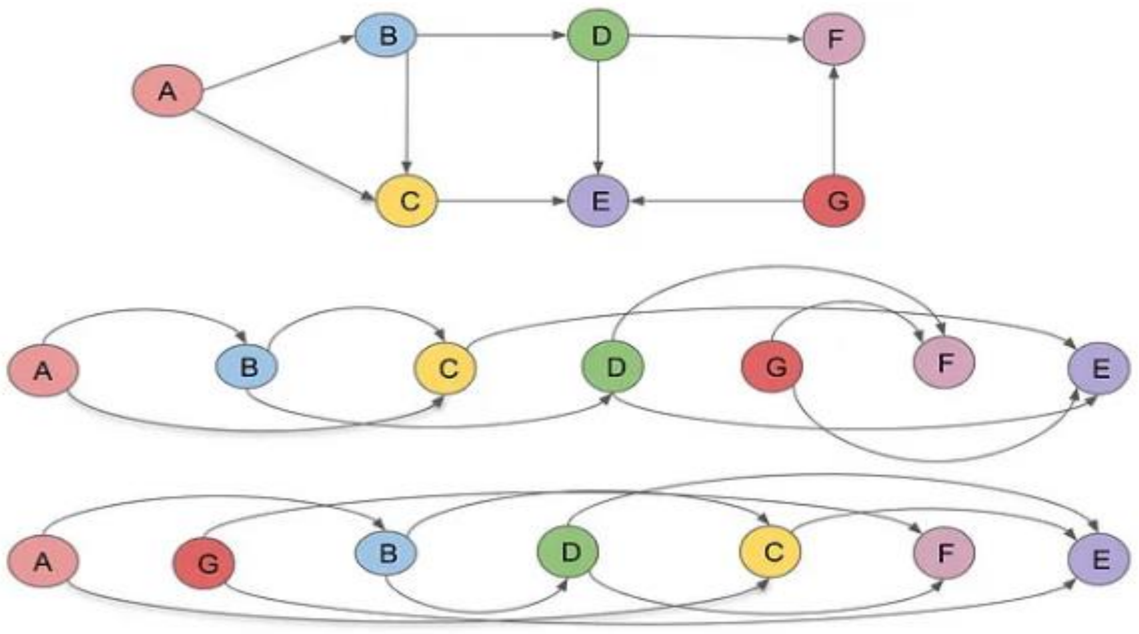


- So [1, 2, 3, 4, 5] would be a topological ordering of the graph.

Algorithm 1: Kahn's Algorithm for Topological Sort

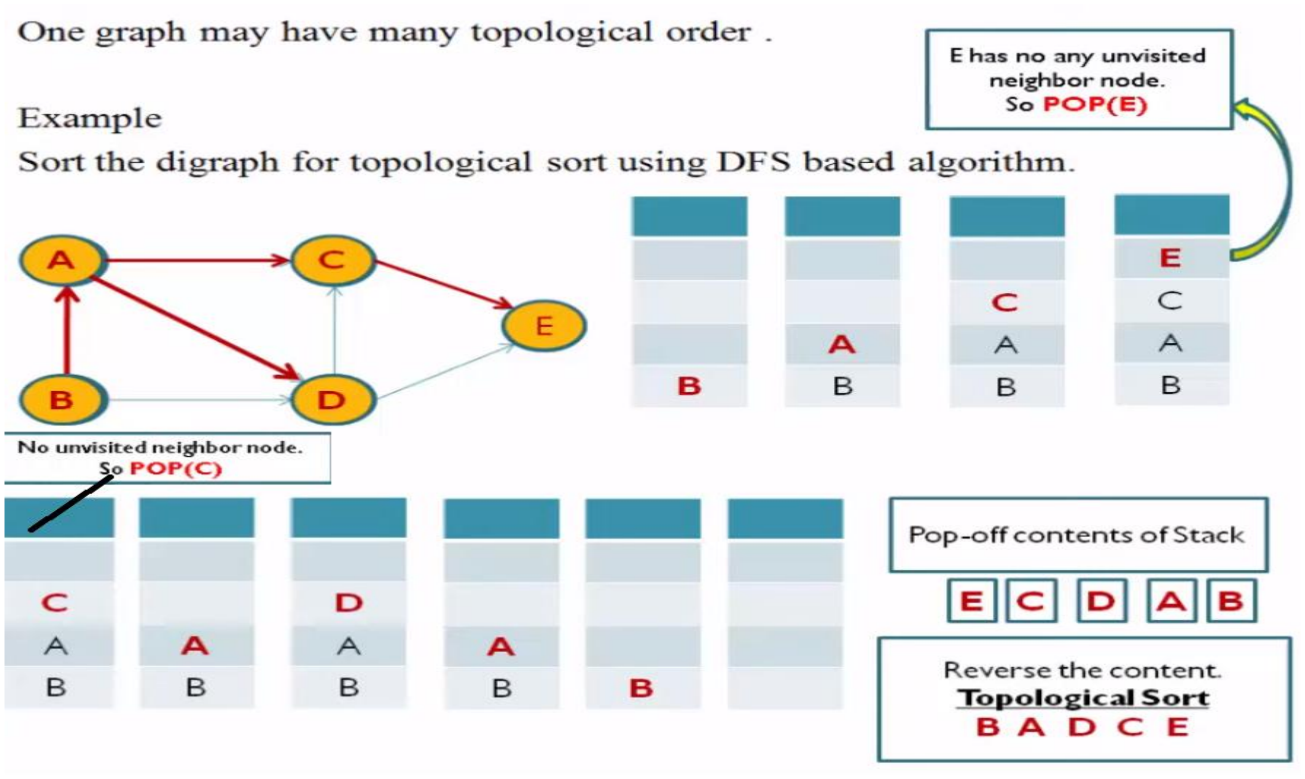
```
Data: A DAG $G$
Result: A topological sort of all vertices in $G$
Compute in-degree (number of incoming edges) for each vertex in $G$;
Put all the vertices with 0 in-degree into a queue $Q$;
Create an empty vertex list $L$;
while $Q$ is not empty do
    Remove a vertex $u$ from $Q$;
    Add $u$ to the end of the $L$;
    foreach $u$'s neighboring node $v$ do
        Decrease $v$'s in-degree by 1;
        if $v$'s in-degree is 0 then
            Add $v$ to $Q$;
        end
    end
end
return $L$;
```

Example:



Topological Sort

Implementation using DFS:


One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.

E has no any unvisited neighbor node. So POP(E)

No unvisited neighbor node. So POP(C)

Pop-off contents of Stack

E C D A B

Reverse the content.

Topological Sort

B A D C E

1. **Parallel Courses:**

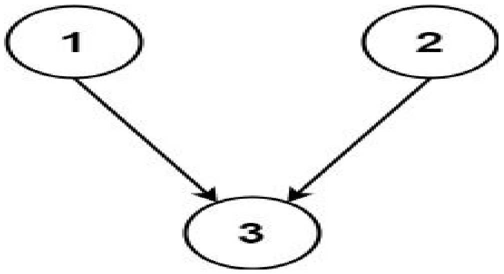
1.1.Parallel Courses-I:

You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given an array relations where relations[i] = [prevCoursei, nextCoursei], representing a prerequisite relationship between course prevCoursei and course nextCoursei: course prevCoursei has to be taken before course nextCoursei.

In one semester, you can take any number of courses as long as you have taken all the prerequisites in the previous semester for the courses you are taking.

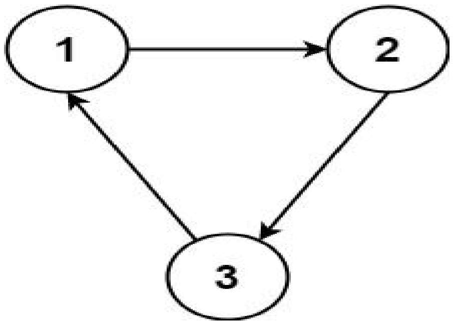
Return the minimum number of semesters needed to take all courses. If there is no way to take all the courses, return -1.

Example 1:



Input: n = 3, relations = [[1,3],[2,3]]
Output: 2
Explanation: The figure above represents the given graph.
In the first semester, you can take courses 1 and 2.
In the second semester, you can take course 3.

Example 2:



Input: n = 3, relations = [[1,2],[2,3],[3,1]]
Output: -1
Explanation: No course can be studied because they are prerequisites of each other.

Solution:

We can first build a graph gg to represent the prerequisite relationships between courses, and count the in-degree $indeg$ of each course.

Then we enqueue the courses with an in-degree of 0 and start topological sorting. Each time, we dequeue a course from the queue, reduce the in-degree of the courses that it points to by 1, and if the in-degree becomes 0 after reduction, we enqueue that course. When the queue is empty, if there are still courses that have not been completed, it means that it is impossible to complete all courses, so we return −1. Otherwise, we return the number of semesters required to complete all courses.

The time complexity is $O(n+m)$, and the space complexity is $O(n+m)$. Here, n and m are the number of courses and the number of prerequisite relationships, respectively.

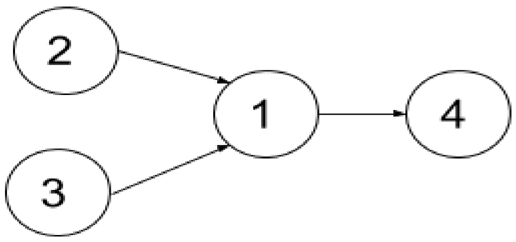
<u>Java program for Implementing Parallel Course Application:</u> **ParallelCourses_I.java**

```
class ParallelCourses_I
 {
   public int minimumSemesters(int n, int[][] relations)
  {
        List<Integer>[] g = new List[n];
        Arrays.setAll(g, k -> new ArrayList<>());
        int[] indeg = new int[n];
        for (var r : relations) {
            int prev = r[0] - 1, nxt = r[1] - 1;
            g[prev].add(nxt);
            ++indeg[nxt];
        }
        Deque<Integer> q = new ArrayDeque<>();
        for (int i = 0; i < n; ++i)
       {
            if (indeg[i] == 0)
          {
                q.offer(i);
            }
        }
        int ans = 0;
        while (!q.isEmpty())
        {
            ++ans;
            for (int k = q.size(); k > 0; --k)
                 {
                int i = q.poll();
                --n;
                for (int j : g[i])
                {
                   if (--indeg[j] == 0)
                     {
                       q.offer(j);
                   }
                }
            }
        }
        return n == 0 ? ans : -1;
    }
}
```

1.2.Parallel Courses-II:

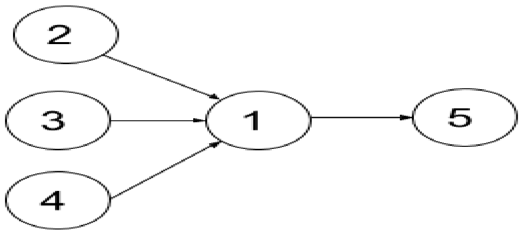
- You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given an array relations where relations[i] = [prevCourse$_i$, nextCourse$_i$], representing a prerequisite relationship between course prevCourse$_i$ and course nextCourse$_i$: course prevCourse$_i$ has to be taken before course nextCourse$_i$. Also, you are given the integer k.
- In one semester, you can take **at most** k courses as long as you have taken all the prerequisites in the **previous** semesters for the courses you are taking.
- Return *the **minimum** number of semesters needed to take all courses*. The testcases will be generated such that it is possible to take every course.

Example 1:



Input: n = 4, relations = [[2,1],[3,1],[1,4]], k = 2
Output: 3
Explanation: The figure above represents the given graph.
In the first semester, you can take courses 2 and 3.
In the second semester, you can take course 1.
In the third semester, you can take course 4.

Example 2:



Input: n = 5, relations = [[2,1],[3,1],[4,1],[1,5]], k = 2
Output: 4
Explanation: The figure above represents the given graph.
In the first semester, you can only take courses 2 and 3 since you cannot take more than two per semester.
In the second semester, you can take course 4.
In the third semester, you can take course 1.

In the fourth semester, you can take course 5.

<u>Java program for Implementing Parallel Course Application:</u> **ParallelCourses_II.java**

```
import java.util.*;
public class ParallelCourses_II
{
public int minimumSemesters(int numCourses, int[][] prerequisites,int maxCourses)
{
      // create an adjacency list to represent the graph
                  int graph[][]=new int[numCourses][numCourses];
                  int[] indegree = new int[numCourses];

      // populate the adjacency list using the prerequisites array

         for (int[] prerequisite : prerequisites)
         {
         int u = prerequisite[0];
         int v = prerequisite[1];
         graph[u][v]=1;
         indegree[v]++;
         }
 // Perform a topological sort to find the order in which the courses should be taken
      Queue<Integer> queue = new LinkedList<>();
      for (int i = 0; i < numCourses; i++)
         {
         if (indegree[i] == 0)
           {
              queue.offer(i);
            }
         }
       int semesters = 0;
      int coursesTaken = 0;
      while (!queue.isEmpty())
           {
         int coursesThisSemester = Math.min(queue.size(), maxCourses);
         for (int i = 0; i < coursesThisSemester; i++)
         {
            int u = queue.poll();
            coursesTaken++;
            for (int v=0;v<numCourses;v++)
```

```
                {
                    if(graph[u][v]==1&&--indegree[v] == 0)
                      queue.offer(v);
                }
            }
            semesters++;
        }
        if (coursesTaken != numCourses)
       {
            return -1; // cannot complete all courses
        }
        return semesters;
      }
            public static void main(String[] args)
            {           Scanner s=new Scanner(System.in);
                        int numCourses=s.nextInt();
                        int c=s.nextInt();
                        int prerequisites[][]=new int[c][2];
                        for(int i=0;i<c;i++)
                        {
                                    for(int j=0;j<2;j++)
                                    {
                                                prerequisites[i][j]=s.nextInt();
                                    }
                        }
                        int maxCourses=s.nextInt();
                    ParallelCourses_II  p=new ParallelCourses_II ();
 System.out.println(p.minimumSemesters(numCourses,prerequisites,maxCourses));
            }
}
```

Input: 4
2 1
3 1
1 4
2
Output: 3

2. Course Schedule:

- There are n courses — course 0 to n-1. Some of them have prerequisites. For example, courses A and B must be completed before course C can be taken (in other words, course C depends on A and B).
- Find and return an order in which all the given courses can be taken, while satisfying all the prerequisites. If there exists more than one such order, any one of them would be a correct answer. If no such order exists, return a special value: [-1].

Example:

Input: n=4, prerequisites=[[1, 0], [2, 0], [3, 1], [3, 2]]

Output: [0, 1, 2, 3]

According to the input:

- Course 0 must be done before both 1 and 2
- Courses 1 and 2 must be done before course 3

There are two orders in which one can take all four courses: [0, 1, 2, 3] and

[0, 2, 1, 3]. Both are correct answers.

Notes:

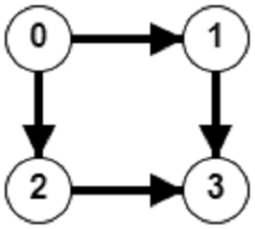
- Function accepts two arguments: The number of courses n and the list of prerequisites.
- Prerequisites are given in the form of a two-dimensional array (or a list of lists) of integers. Each inner array has exactly two elements — it is essentially a list of pairs. Each pair [X, Y] represents one prerequisite: Course Y must be completed before X X depends on Y).
- Function must return an array (list) of integers.
- If all given courses can be taken while satisfying all given prerequisites, the returned array must contain a possible ordering (if more than one such ordering exists, any one must be returned). Otherwise, the function must return an array (list) with one element -1 in it.

Approach:

- This is a Topological Sort problem. We'll cover two efficient sample solutions: one uses DFS, while the second keeps track of the in-degree of the graph nodes. They have the same time and space complexity in the Big-O notation in terms of the number of courses and the number of prerequisites.
- Both algorithms need a directed graph to work with. So, let us build one.

a. Courses become graph nodes.
b. Prerequisites become directed edges: for each "course A must be completed before B can be taken," we add an edge from A->B. In other words, an incoming edge means that another course must be completed before this one.

For example, if n=4 and prerequisites= [[1, 0], [2, 0], [3, 1], [3, 2]], the graph would look like this:



- ➢ The correct answer to the problem will be a topological order of the nodes. The two sample solutions discussed below are essentially two different implementations of the topological sort algorithm.
- ➢ Topological ordering doesn't exist if the graph has a cycle. Both sample solutions detect cycles in their respective ways and return the special value in that case.

<u>Java Program for Implementing Course Schedule Application</u>: CourseSchedule.java

```
import java.util.*;

public class CourseSchedule
{
   public static boolean canFinish(int numCourses, int[][] prerequisites)
   {
      int adjList[][]=new int[numCourses][numCourses];

      // Populate the adjacency list with prerequisites
      int[] inDegree = new int[numCourses];
      for (int[] prerequisite : prerequisites)
                  {
         int u = prerequisite[0];
         int v = prerequisite[1];
         adjList[v][u]=1;
         inDegree[u]++;
      }
    // Perform a topological sort to find the order in which the courses should be taken
      Queue<Integer> queue = new LinkedList<>();
      for (int i = 0; i < numCourses; i++)
      {
         if (inDegree[i] == 0)
         {
             queue.offer(i);
         }
      }
      int count = 0;
      int[] order = new int[numCourses];
      int index = 0;
      while (!queue.isEmpty())
      {
         int u = queue.poll();
         count++;
          order[index++] = u;
         for (int v=0;v<numCourses;v++)
          {
             if (adjList[u][v]==1&&--inDegree[v] == 0)
             {
                 queue.offer(v);
             }
         }
      }
      if(count == numCourses)
      {
         for(int i=0;i<count;i++)
         System.out.print(order[i]+" ");
         return true;
      }
     else
     {
         return false;
     }
   }
   public static void main(String[] args)
   {
         Scanner s=new Scanner(System.in);
         int nc=s.nextInt();
         int c=s.nextInt();
         int prereq[][]=new int[c][2];
         for(int i=0;i<c;i++)
         {
                 for(int j=0;j<2;j++)
                 {
                 prereq[i][j]=s.nextInt();
                 }
         }
         System.out.println(canFinish(nc, prereq));
         }
 }
```

```
Case=1
  Input=5
    4
    2 4
    3 1
    0 2
    4 3
 Output=1 3 4 2 0 true

 Case=2
 Input=4
   4
  1 0
  2 0
  3 1
  3 2
Output =0 1 2 3 true
```