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:

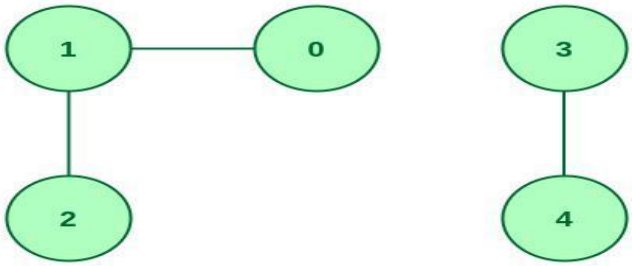
Applications:

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 ∈ V do
    Visited[k] = False;
end
for each vertex k ∈ 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 ∈ V.Adj[k] do
    if Visited[p] == False then
        DFS(V,p);
    end
end
```

Time Complexity: O(V+E)

Where:

- V= number of vertices (nodes),
- E = number of edges.

Explanation:

- You visit **every vertex once** (that's O(V)),
- You explore **every edge once** (that's O(E),
- DFS takes linear time in the size of the graph (vertices + edges combined).

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
```

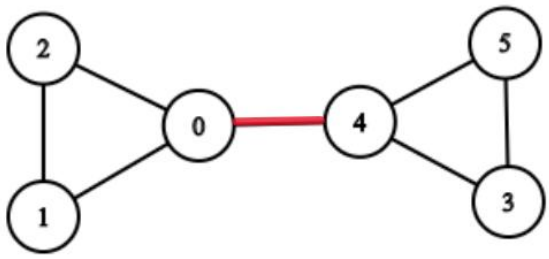
```
                // adjacent to this vertex
                for (int x : adjListArray.get(v))
                    {
                        if (!visited[x])
                                DFSUtil(x, visited);
                }
        }
        void connectedComponents()
        {
                // 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 :



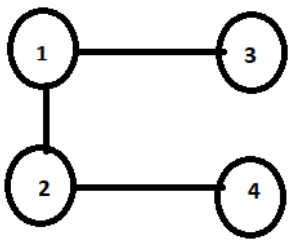
- The edge between 0 and 4 is the bridge because if the edge between 0 and 4 is removed, then there will be no path left to reach from 0 to 4.and makes the graph disconnected, and increases the number of connected components.
- **Note :**
  - There are no self-loops(an edge connecting the vertex to itself)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

Pick an arbitrary vertex of the graph root x and run depth first search from it. Note the following fact (which is easy to prove):

- Let's say we are in the DFS, looking through the edges starting from vertex v.The current edge (v,to) is a bridge if and only if none of the vertices to and its descendants in the DFS traversal tree has a back-edge to vertex v $ or any of its ancestors. Indeed, this condition means that there is no other way from v<math to to except for edge (v,to).

Now we have to learn to check this fact for each vertex efficiently. We'll use "time of entry into node" computed by the depth first search.

So, let tin[v] denote entry time for node v<. We introduce an array lowt=" which will let us store the node with earliest entry time found in the DFS search that a node v can reach with a single edge from itself or its descendants. low[v]v]$ is the minimum of tin[v]<, the entry times tin[p]< for each node p<math $ that is connected to node v< via a back-edge (v,p)< and the values of low[to] for each vertex to which is a direct descendant of v< in the DFS tree:

$$\texttt{low}[v] = \min \begin{Bmatrix} \texttt{tin}[v] & \\ \texttt{tin}[p] & \text{for all } p \text{ for which } (v,p) \text{ is a back edge} \\ \texttt{low}[to] & \text{for all } to \text{ for which } (v,to) \text{ is a tree edge} \end{Bmatrix}$$

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which $\texttt{low}[to] \le \texttt{tin}[v]$. If $\texttt{low}[to] = \texttt{tin}[v]$, the back edge comes directly to v, otherwise it comes to one of the ancestors of v.

Thus, the current edge (v, to) in the DFS tree is a bridge if and only if $\texttt{low}[to] > \texttt{tin}[v]$.

Implementation

The implementation needs to distinguish three cases: when we go down the edge in DFS tree, when we find a back edge to an ancestor of the vertex and when we return to a parent of the vertex. These are the cases:

- $\texttt{visited}[to] = false$ - the edge is part of DFS tree;
- $\texttt{visited}[to] = true$ && $to \neq parent$ - the edge is back edge to one of the ancestors;
- $to = parent$ - the edge leads back to parent in DFS tree.

To implement this, we need a depth first search function which accepts the parent vertex of the current node.

For the cases of multiple edges, we need to be careful when ignoring the edge from the parent. To solve this issue, we can add a flag `parent_skipped` which will ensure we only skip the parent once.

Time Complexity: O(V+E)

Where:

- V= number of vertices (nodes),
- E = number of edges.

Explanation:

- It does a single **DFS traversal** (visits each vertex once, O(V)),
- It processes each edge exactly **once** (checking and updating discovery and low values, O(E)),
- So total work = O(V+E).
-

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
```

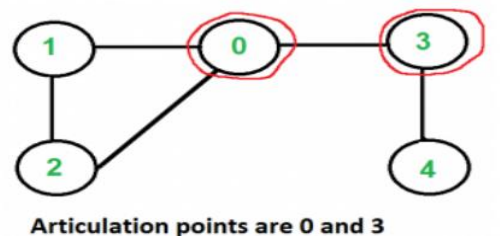
3. Finding Articulation Point in a Graph:

- 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}



Output = 0,3

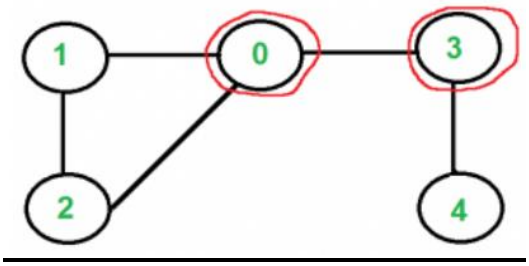
Algorithm:

Algorithm 1: Algorithm to Find all the Articulation Points

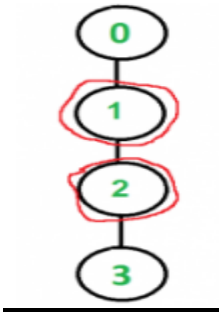
```
Data: G(V, E), s
Result: Articulation vertices in G
DFSTree(G);
Procedure ArticulationPoints(s, d)
Visited[s] = TRUE;
depth[s] = d;
low[s] = d;
for each k ∈ Adj(s) do
    if Visited[s] == FALSE then
        ArticulationPoint(k, d+1);
    end
    low[s] = min (low[s], low[k]);
    if low[k] ≥ depth[s] 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.

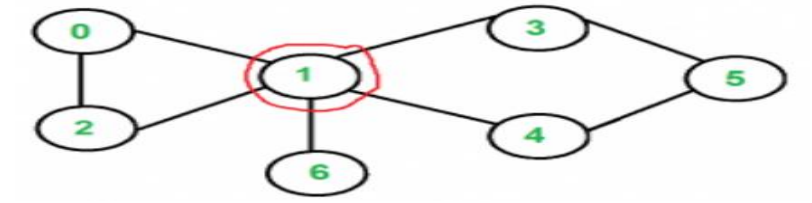
Graph 1: (Articulation Point Graph 1: 0,3)



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



Graph 3: (Articulation Point Graph 3: 1)



Time Complexity: O(V+E)

Where:

- V= number of vertices (nodes),
- E = number of edges.

Explanation:

- It does a single **DFS traversal** (visits each vertex once, O(V)),
- It processes each edge exactly **once** (checking and updating discovery and low values, O(E)),
- So total work = O(V+E).
-

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 Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method for computing a maximal flow in a flow network.

Flow network: First let's define what a flow network**, a** flow**, and a** maximum flow **is.**

A **network** is a directed graph G with vertices V and edges E combined with a function c, which assigns each edge e∈E< a non-negative integer value, the **capacity** of e. Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and one as **sink**.

A **flow** in a flow network is function f >f, that again assigns each edge ea non-negative integer value, namely the flow. The function has to fulfill the following two conditions:

The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

And the sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u except in the source and sink vertices.

$$\sum_{(v,u) \in E} f((v,u)) = \sum_{(u,v) \in E} f((u,v))$$

The source vertex s only has an outgoing flow, and the sink vertex t has only incoming flow.

It is easy to see that the following equation holds:

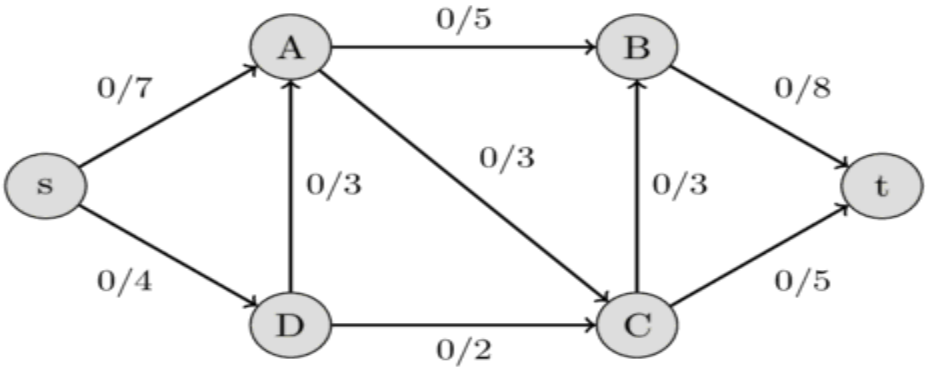
$$\sum_{(s,u) \in E} f((s,u)) = \sum_{(u,t) \in E} f((u,t))$$

A good analogy for a flow network is the following visualization:

- We represent edges as water pipes, the capacity of an edge is the maximal amount of water that can flow through the pipe per second, and the flow of an edge is the amount of water that currently flows through the pipe per second.
- This motivates the first flow condition. There cannot flow more water through a pipe than its capacity.
- The vertices act as junctions, where water comes out of some pipes, and then, these vertices distribute the water in some way to other pipes.

- This also motivates the second flow condition. All the incoming water has to be distributed to the other pipes in each junction. It cannot magically disappear or appear. The source **s** is origin of all the water, and the water can only drain in the sink **t**.

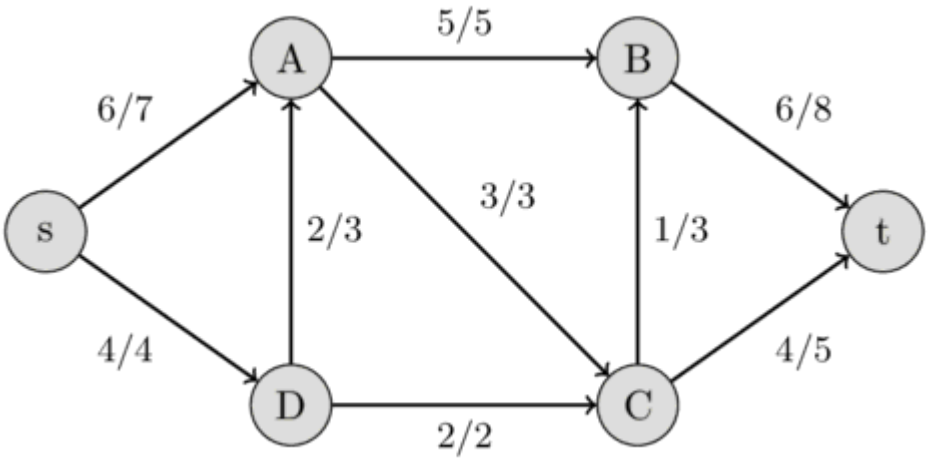
The following image shows a flow network. The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



The value of the flow of a network is the sum of all the flows that get produced in the source s, or equivalently to the sum of all the flows that are consumed by the sink t. A **maximal flow** is a flow with the maximal possible value. Finding this maximal flow of a flow network is the problem that we want to solve.

In the visualization with water pipes, the problem can be formulated in the following way: how much water can we push through the pipes from the source to the sink?

The following image shows the maximal flow in the flow network.



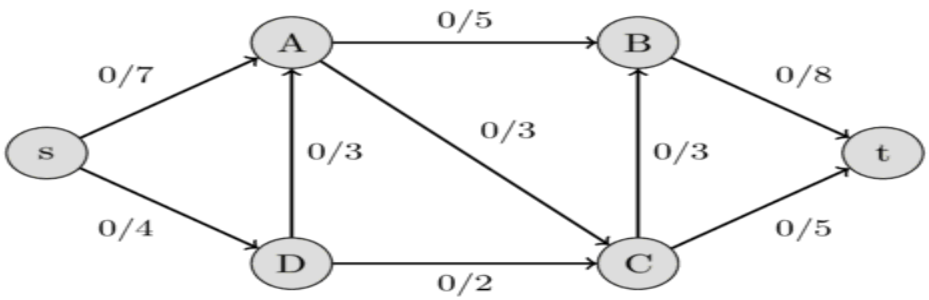
Ford-Fulkerson method:

Let's define one more thing. A **residual capacity** of a directed edge is the capacity minus the flow. It should be noted that if there is a flow along some directed edge **(u,v)** then the reversed edge has capacity 0 and we can define the flow of it as **f(u,v) = -f((u,v)).** This also defines the residual capacity for all the reversed edges. We can create a **residual network** from all these edges, which is just a network with the same vertices and edges, but we use the residual capacities as capacities.

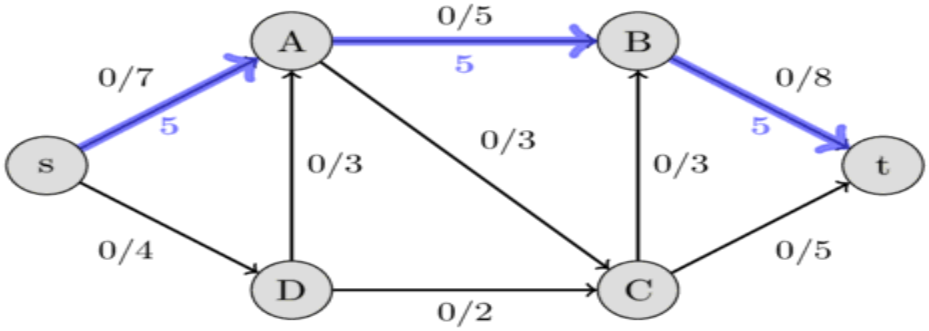
The Ford-Fulkerson method works as follows. First, we set the flow of each edge to zero. Then we look for an **augmenting path** from s to t. An augmenting path is a simple path in the residual graph where residual capacity is positive for all the edges along that path. If such a path is found, then we can increase the flow along these edges. We keep on searching for augmenting paths and increasing the flow. Once an augmenting path doesn't exist anymore, the flow is maximal.

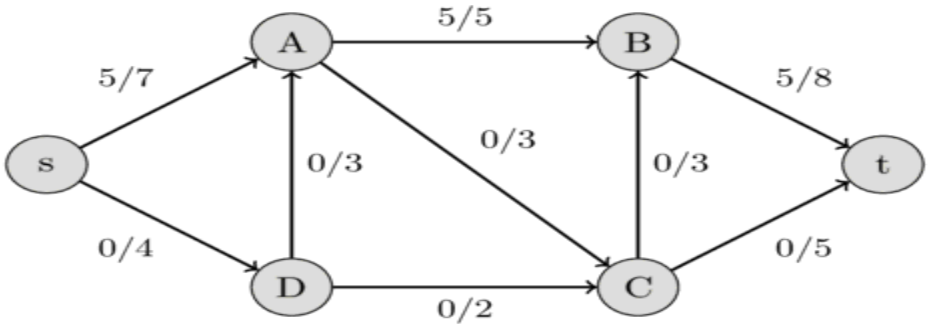
Let us specify in more detail, what increasing the flow along an augmenting path means. Let **C** be the smallest residual capacity of the edges in the path. Then we increase the flow in the following way: we update **f(u,v) += C and f((v,u)) - = C** for every edge **(u,v)** in the path.

Here is an **Example-1** to demonstrate the method. We use the same flow network as above. Initially we start with a flow of 0.

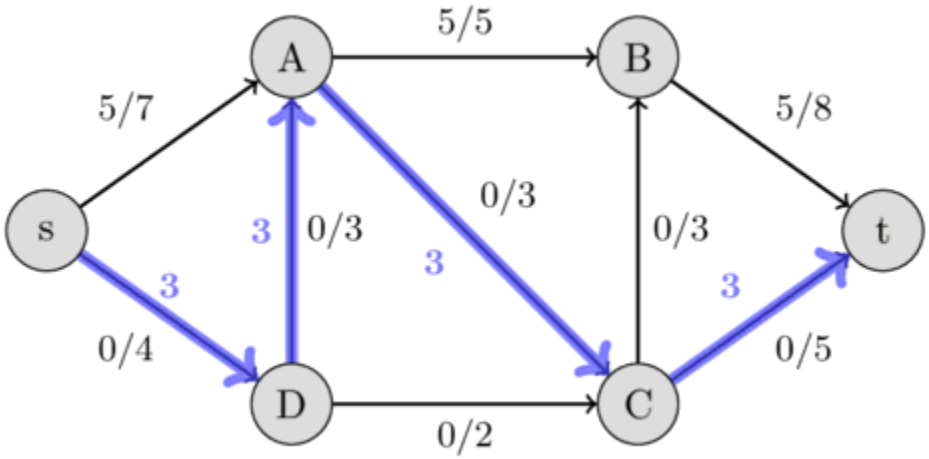


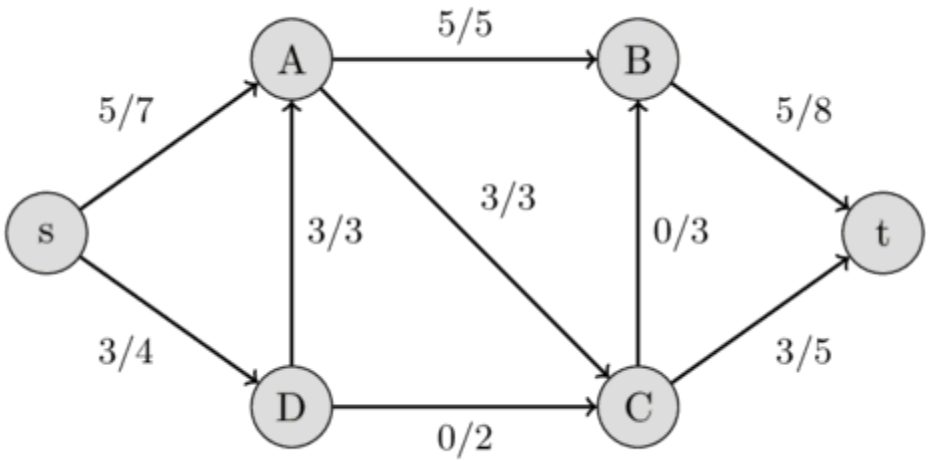
We can find the path **s−A−B−t** with the residual capacities 7, 5, and 8. Their minimum is 5, therefore we can increase the flow along this path by 5. This gives a flow of 5 for the network.



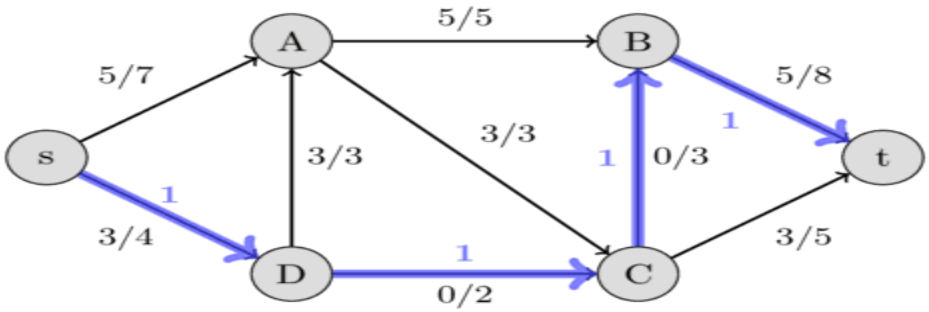


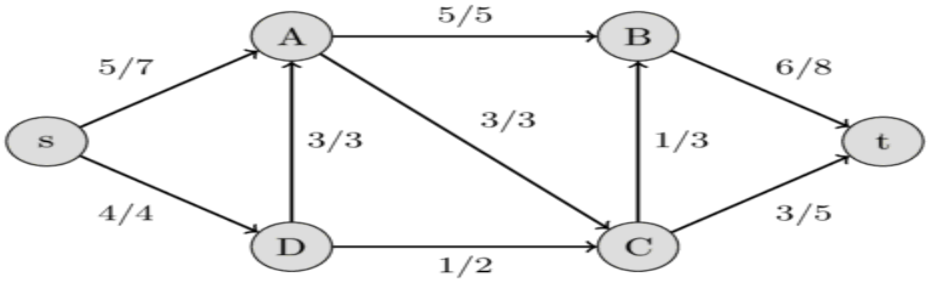
Again we look for an augmenting path, this time we find **s−D−A−C−t** with the residual capacities 4, 3, 3, and 5. Therefore we can increase the flow by 3 and we get a flow of 8 for the network.





This time we find the path **s−D−C−B−t** with the residual capacities 1, 2, 3, and 3, and hence, we increase the flow by 1.

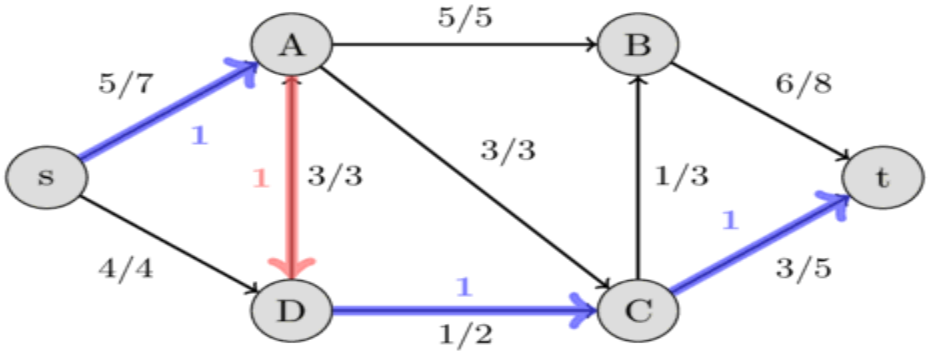


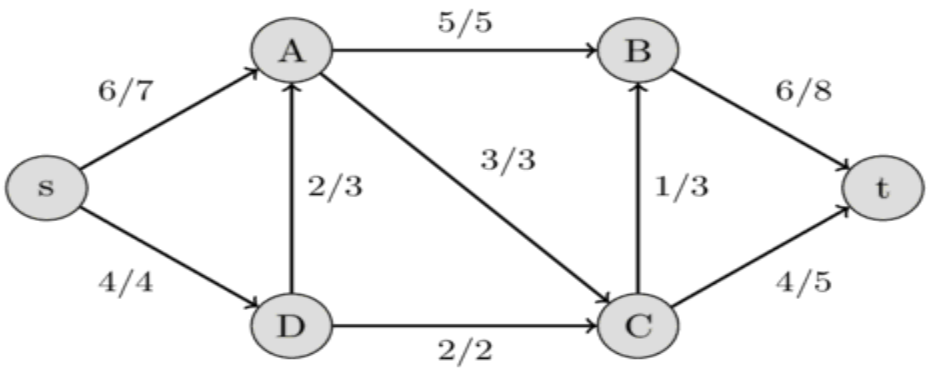


This time we find the augmenting path **s−A−D−C−t** with the residual capacities 2, 3, 1, and 2. We can increase the flow by 1.

But this path is very interesting. It includes the reversed edge **(A,D).** In the original flow network, we are not allowed to send any flow from **A to D.** But because we already have a flow of 3 from **D to A**, this is possible.

The intuition of it is the following: Instead of sending a flow of 3 from **D to A** we only send 2 and compensate this by sending an additional flow of 1 from s to A, which allows us to send an additional flow of 1 along the path **D−C−t.**





Now, it is impossible to find an augmenting path between **s** and **t**, therefore this flow of 10 is the maximal possible. We have found the maximal flow.

It should be noted, that the Ford-Fulkerson method doesn't specify a method of finding the augmenting path. Possible approaches are using DFS or BFS which both work in **O(E)**.

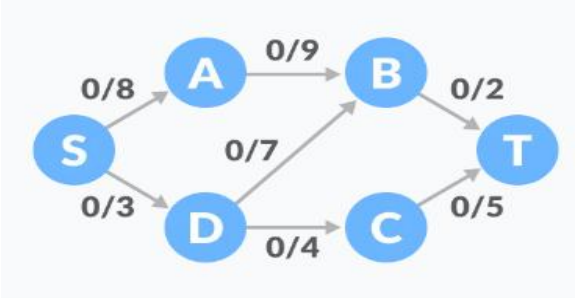
If all the capacities of the network are integers, then for each augmenting path the flow of the network increases by at least 1 (for more details see Integral flow theorem).

Therefore, the complexity of Ford-Fulkerson is **O(E*F)** where **F** is the maximal flow of the network.

In the case of rational capacities, the algorithm will also terminate, but the complexity is not bounded. In the case of irrational capacities, the algorithm might never terminate, and might not even converge to the maximal flow.

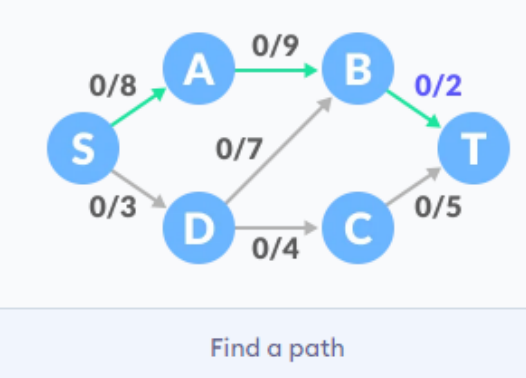
Ford-Fulkerson Example-2

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



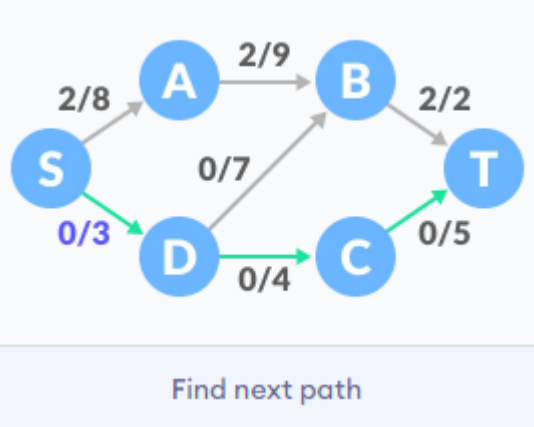
Flow network graph example

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

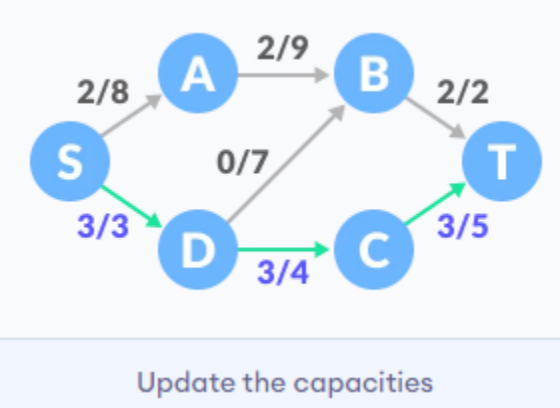


The minimum capacity among the three edges is 2 (B-T) Based on this, update the flow/capacity for each path.

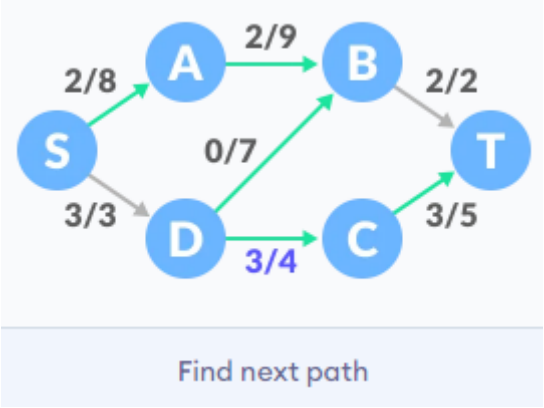
Step2: Select another path **S-D-C-T**. The minimum capacity among these edges is 3**(S-D)**



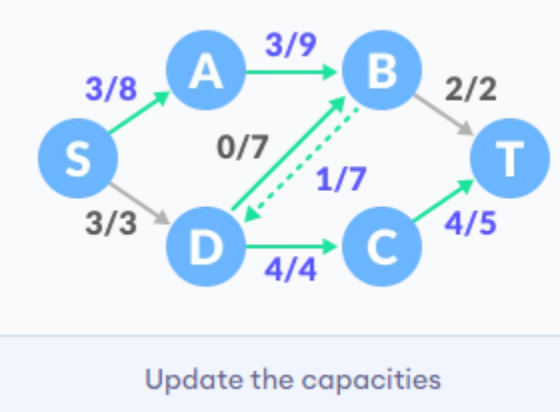
Update the capacities according to this.



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).**

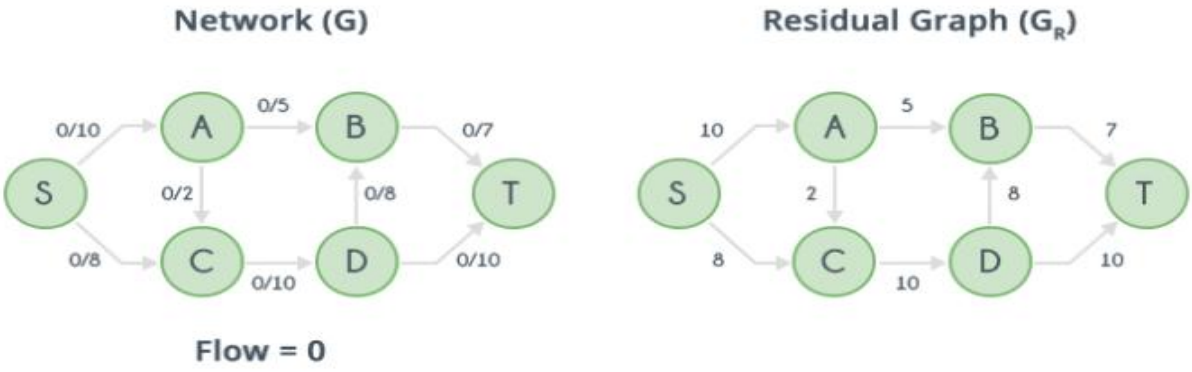


Updating the capacities.

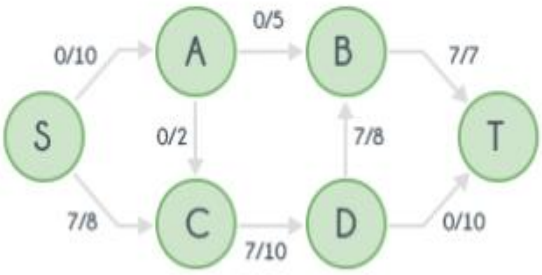


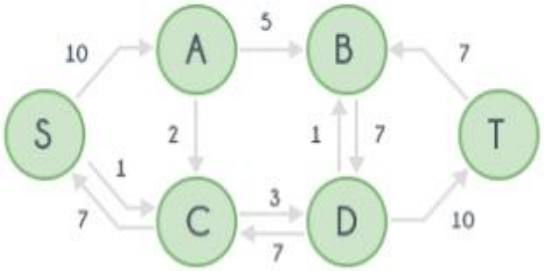
Step 4: Adding all the flows = 2 + 3 + 1 = 6, which is the maximum possible flow on the flow network.

Example 3:

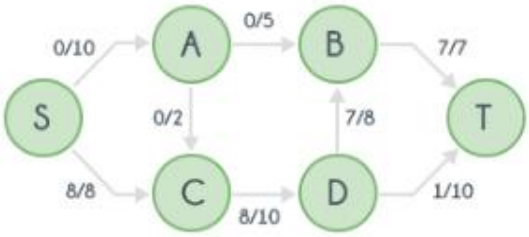


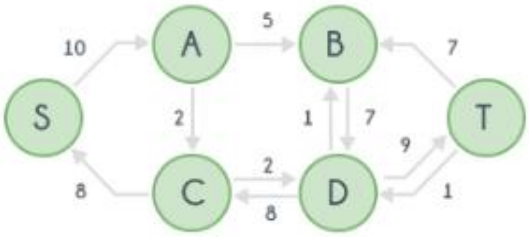
Path 1: S - C - D - B - T ⟶ Flow = Flow + 7



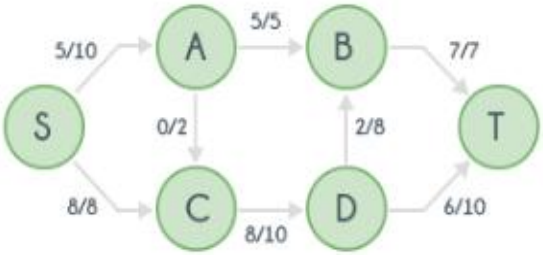


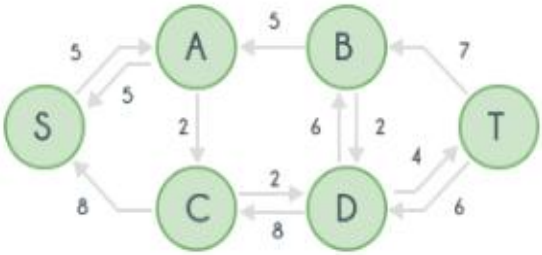
Path 2: S - C - D - T ⟶ Flow = Flow + 1

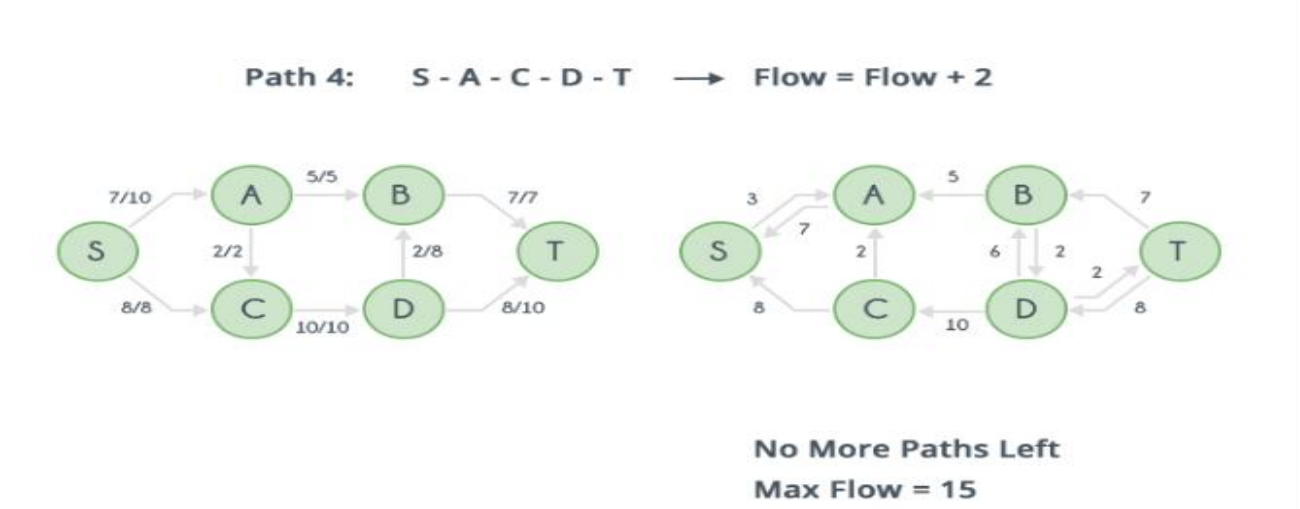




Path 3: S - A - B - T ⟶ Flow = Flow + 5







Steps to Solve Max Flow using DFS (Ford-Fulkerson Algorithm)

1. Represent the Graph

- Represent the network as a **capacity graph** (typically as an adjacency matrix or adjacency list).
- Let the graph have V vertices, a **source** node S, and a **sink** node T.

2. Create the Residual Graph

- Initialize a **residual graph** from the original graph, which stores the remaining capacity of each edge.
- Residual capacity of an edge = original capacity − current flow.

3. Initialize the Flow

- Set maxFlow = 0.

4. Search for Augmenting Paths (Using DFS)

- While a path exists from S to T in the residual graph:
  - Use **DFS** to find a path from S to T such that all edges in the path have **residual capacity > 0**.
  - If no such path exists, terminate.

5. Determine Bottleneck Capacity

- For the path found by DFS:
  - Determine the **minimum residual capacity** (this is the maximum flow you can push through this path).
  - This is called the **bottleneck capacity** or **path flow**.

6. Update Residual Capacities

- For each edge (u, v) in the path:
  - Decrease residual[u][v] by path_flow
  - Increase residual[v][u] by path_flow (this is the reverse edge to allow flow cancellation)

7. Update Total Flow

- Add the **path flow** to maxFlow.

8. Repeat Until No Augmenting Path Exists

- Go back to Step 4.
- Stop when DFS can no longer find a path from S to T in the residual graph.

9. Output the Maximum Flow

- The maxFlow at this point is the **maximum flow from source to sink**.

Java program for implementation of Ford Fulkerson algorithm using DFS:

MaxFlowFordFulkerson_DFS.java

```
import java.util.*;

public class MaxFlowFordFulkerson_DFS
 {

   static  int V ; // Number of vertices (you can change this dynamically if needed)

   // Returns true if there is a path from source to sink in residual graph
   // and fills parent[] to store the path
   boolean dfs(int[][] rGraph, int s, int t, int[] parent)
   {
      boolean[] visited = new boolean[V];
      Stack<Integer> stack = new Stack<>();
      stack.push(s);
      visited[s] = true;
      parent[s] = -1;

      while (!stack.isEmpty())
      {
         int u = stack.pop();

         for (int v = 0; v < V; v++)
         {
            if (!visited[v] && rGraph[u][v] > 0)
            {
               stack.push(v);
               parent[v] = u;
               visited[v] = true;
               if (v == t) return true;
            }
         }
      }
      return false;
   }

   // Returns the maximum flow from s to t in the given graph
```

```
int fordFulkerson(int[][] graph, int s, int t)
{
   int u, v;

   // Create residual graph
   int[][] rGraph = new int[V][V];
   for (u = 0; u < V; u++)
      for (v = 0; v < V; v++)
         rGraph[u][v] = graph[u][v];

   int[] parent = new int[V];
   int maxFlow = 0;

   // Augment the flow while there is a path from source to sink
   while (dfs(rGraph, s, t, parent))
   {
      int pathFlow = Integer.MAX_VALUE;

      // Find minimum residual capacity along the path filled by DFS
      for (v = t; v != s; v = parent[v])
       {
         u = parent[v];
         pathFlow = Math.min(pathFlow, rGraph[u][v]);
      }

      // Update residual capacities
      for (v = t; v != s; v = parent[v]) {
         u = parent[v];
         rGraph[u][v] -= pathFlow;
         rGraph[v][u] += pathFlow;
      }

      // Add path flow to overall flow
      maxFlow += pathFlow;
   }

   return maxFlow;
}

public static void main(String[] args) throws java.lang.Exception
{
   Scanner sc=new Scanner(System.in);
   System.out.println("Enter number of vertices ");
   V= sc.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]=sc.nextInt();
        System.out.println("Enter source and sink ");
        int s = sc.nextInt();
        int t = sc.nextInt();
        System.out.println(new Test().fordFulkerson(graph, s, t));
    }
}
```

Input =

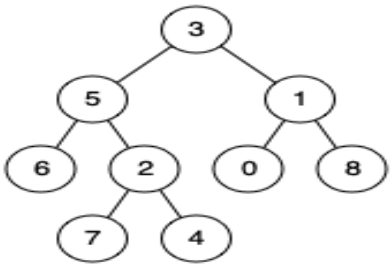
```
Enter number of vertices
6
Enter the adjacency matrix of the directed graph
0 16 13  0   0   0
0  0  10 12  0   0
0  4   0   0  14  0
0  0   9   0   0  20
0  0   0   7   0  4
0  0   0   0   0  0
Enter source and sink
0 5
```

Output = 23

5. 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:-

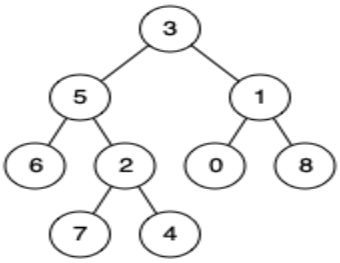


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

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

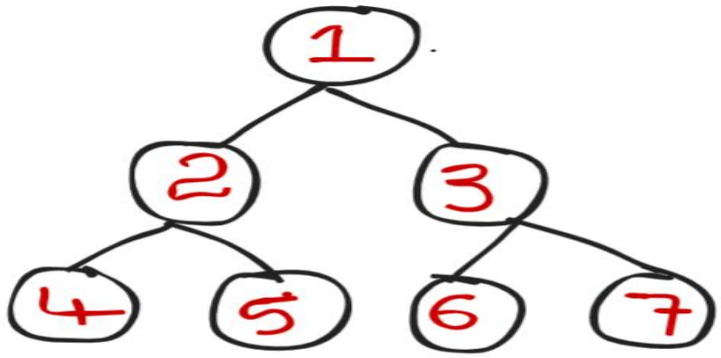
ii)Right subtree

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)

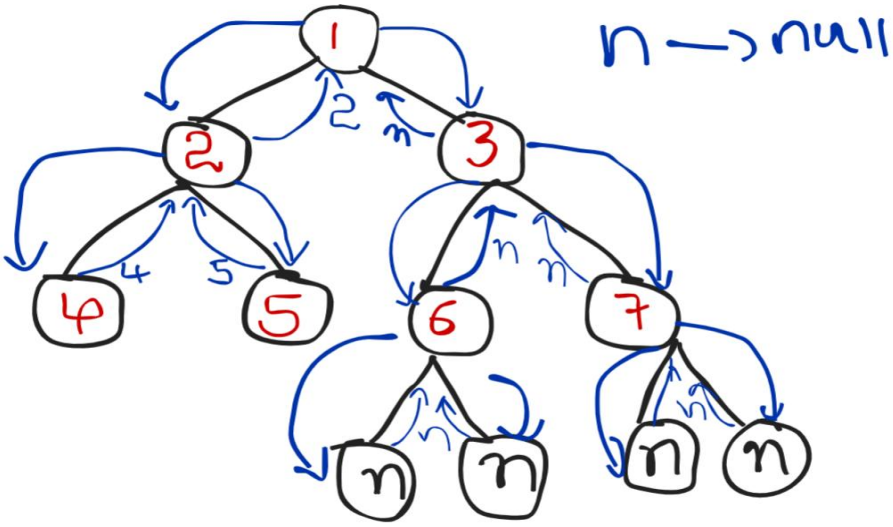
i) Call the right subtree (i.e right of 2), While calling recursively it will find 5 and this call will return 5 to its parent.

- 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

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

```
import java.util.*;

class Node
{
   public int data;
   public Node left;
   public Node right;
   public Node(int value)
 {
       data = value;
       left = null;
       right = null;
   }
}

class Solution
{
   Node lowestCommonAscendant(Node root,Node P1, Node P2){
       if (root == null || root.data == P1.data ||
          root.data == P2.data)

     return root;
       Node left = lowestCommonAscendant(root.left, P1, P2);

       Node right = lowestCommonAscendant(root.right, P1, P2);

       return left == null ? right : right == null ? left : root;
   }
}


public class  Test
{
   public static void main(String[] args)
   {
       Scanner sc = new Scanner(System.in);

       String[] arr= sc.nextLine().split(" ");
       String[] persons = sc.nextLine().split(" ");

       List<Integer> v = new ArrayList<>();

       int n=arr.length;
```

```
        for (int i = 0; i < n; i++)
        {
            v.add(Integer.parseInt(arr[i]));
        }

        Node root = new Node(v.get(0));

        Node P1 = new Node(Integer.parseInt(persons[0]));

        Node P2 = new Node(Integer.parseInt(persons[1]));

        Queue<Node> q = new LinkedList<>();
        Queue<Node> q2 = new LinkedList<>();

        q.add(root);

        int j = 1;
        while (j < n && !q.isEmpty())
         {
            Node temp = q.poll();
            if (v.get(j) != -1)
             {
                temp.left = new Node(v.get(j));
                q.add(temp.left);
            }

            j++;

            if (j < n && v.get(j) != -1)
              {
                temp.right = new Node(v.get(j));
                q.add(temp.right);
            }

            j++;
        }

        Node res=new Solution().lowestCommonAscendant(root, P1, P2);
        System.out.println(res.data);
    }
}
```

Topological Sort:

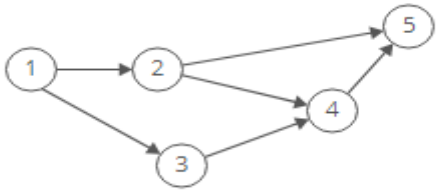
Applications:

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

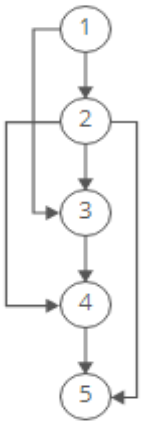
Topological sort:

- Topological Sorting or Kahn's algorithm is an algorithm that orders a Directed acylic graph**(DAG)** 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 1:

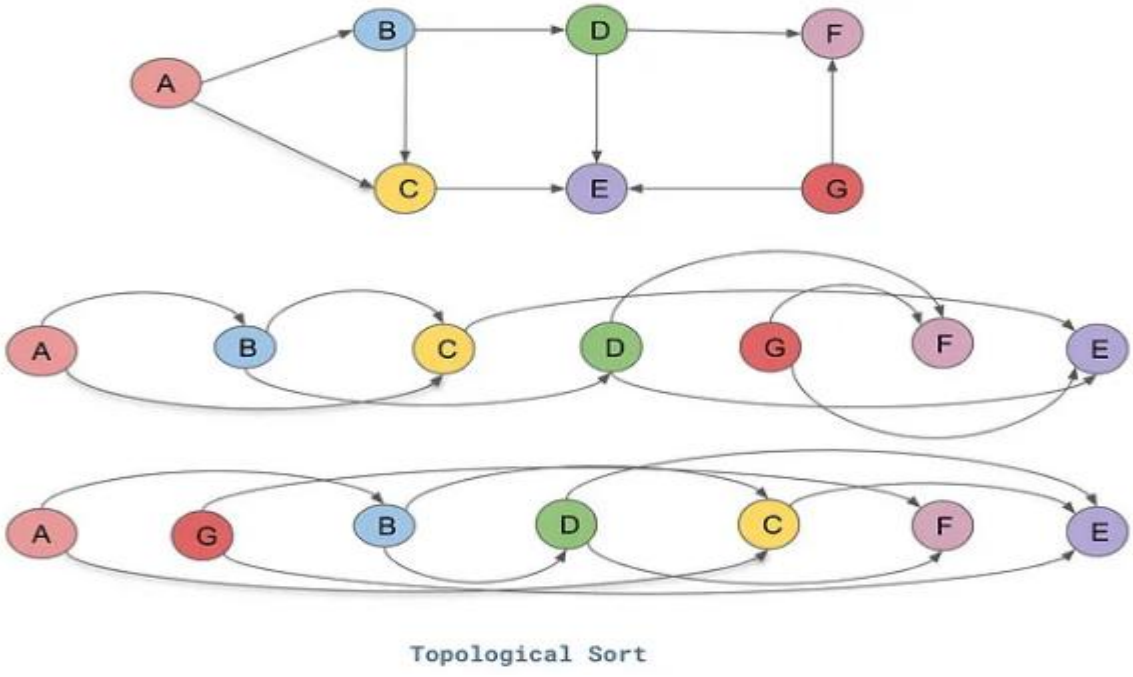


- Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.



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

Example-2:



Topological Sort

Two important methods to solve are:

A. **Kahn's Algorithm (Using Queue Data Struucture)**
B. **Depth-First Search (DFS) Algorithm**

A. Kahn's Algorithm

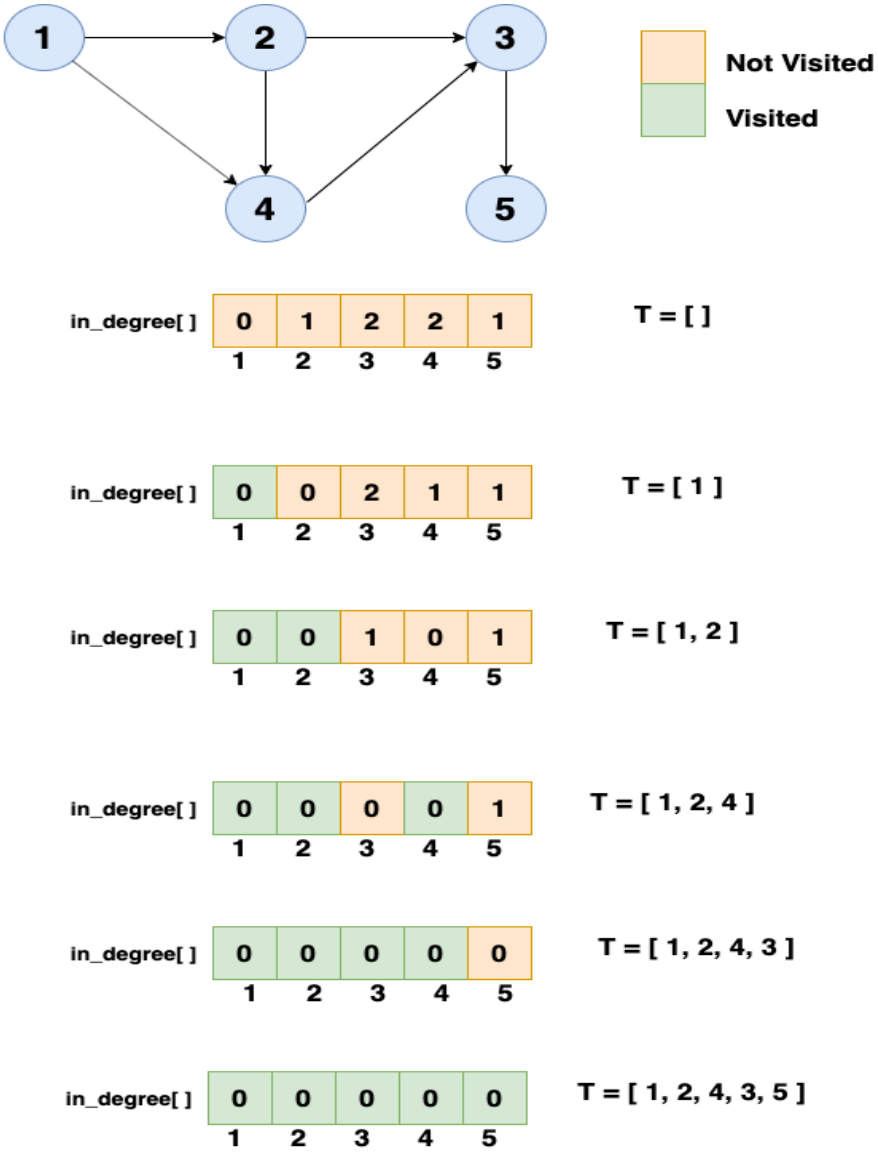
- Kahn's Algorithm is a topological sorting algorithm that uses a queue-based approach to sort vertices in a DAG.
- It starts by finding vertices that have no incoming edges and adds them to a queue.
- It then removes a vertex from the queue and adds it to the sorted list.
- The algorithm continues this process, removing vertices with no incoming edges until all vertices have been sorted.

Procedure:

- For that, we will maintain an array T[ ], which will store the ordering of the vertices in topological order.
- We will store the number of edges that are coming into a vertex in an array in_degree[N], where the i-th element will store the number of edges coming into the vertex i.
- We will also store whether a certain vertex has been visited or not in visited[N].
  We will follow the belo steps:
  - ✓ First, take out the vertex whose in_degree is 0. That means there is no edge that is coming into that vertex.

- ✓ We will append the vertices in the Queue and mark these vertices as visited.
- ✓ Now we will traverse through the queue and in each step we will dequeue() the front element in the Queue and push it into the T.
- ✓ Now, we will put out all the edges that are originated from the front vertex which means we will decrease the in_degree of the vertices which has an edge with the front vertex.
- ✓ Similarly, for those vertices whose in_degree is 0, we will push it in Queue and also mark that vertex as visited. (Hope you must be thinking its BFS but with in_degree).

Illustration



Time Complexity: O(V + E)
Auxiliary Space: O(V)

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;
```

Java program for Topological sort using Queue (Kahn's Algorithm):

TopologicalSort_Queue.java

```
import java.util.*;

class Solution
{
   public List<Integer> topologicalSort(int N, ArrayList<ArrayList<Integer>> adj)
   {
      int[] indegree = new int[N];

      // Compute indegree of each node
      for (int i = 0; i < N; i++)
       {
          for (int node : adj.get(i))
           {
              indegree[node]++;
          }
      }
      Queue<Integer> q = new LinkedList<>();
      // Add all nodes with indegree 0 to the queue
      for (int i = 0; i < N; i++)
       {
          if (indegree[i] == 0)
          {
```

```
            q.add(i);
        }
    }
    List<Integer> topoOrder = new ArrayList<>();
    int count = 0;
    while (!q.isEmpty())
    {
        int node = q.poll();
        topoOrder.add(node);
        count++;
        for (int neighbor : adj.get(node))
        {
            indegree[neighbor]--;
            if (indegree[neighbor] == 0)
            {
                q.add(neighbor);
            }
        }
    }
    // If cycle exists
    if (count != N)
    {
        return new ArrayList<>(); // Empty list indicates a cycle
    }
    return topoOrder;
  }
}
public class TopologicalSort_Queue {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of vertices: ");
    int N = sc.nextInt();

    System.out.print("Enter number of edges: ");
    int E = sc.nextInt();

    ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
    for (int i = 0; i < N; i++) {
        adj.add(new ArrayList<>());
    }
    System.out.println("Enter edges (from to):");
    for (int i = 0; i < E; i++) {
        int u = sc.nextInt();
        int v = sc.nextInt();
```

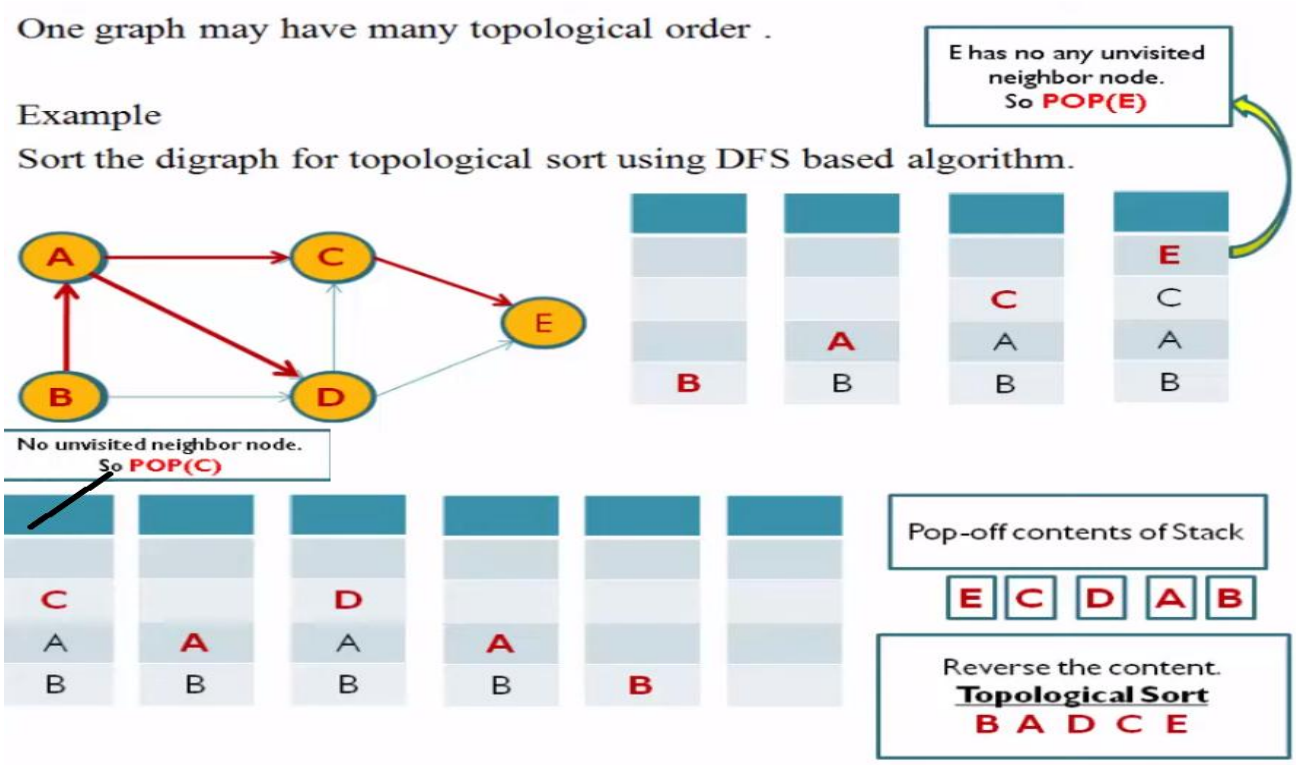
```
            adj.get(u).add(v); // Directed edge
        }
        Solution sol = new Solution();
        List<Integer> result = sol.topologicalSort(N, adj);

        if (result.isEmpty()) {
            System.out.println("Graph contains a cycle. Topological sort not possible.");
        } else {
            System.out.println("Topological order is:");
            for (int node : result) {
                System.out.print(node + " ");
            }
            System.out.println();
        }

        sc.close();
    }
}
```

Example-1:
Input=
Enter number of vertices: 6
Enter number of edges: 9
Enter edges (from to):
5 0
5 2
2 0
2 3
3 0
3 1
1 0
4 0
4 1
Output=
Topological order is:
4 5 2 3 1 0

B. Topological sort implementation using DFS:

Approach:

- DFS Approach is a recursive algorithm that performs a depth-first search on the DAG.
- It starts at a vertex, explores as far as possible along each branch before backtracking, and marks visited vertices.
- During the DFS traversal, vertices are added to a stack in the order they are visited.
- Once the DFS traversal is complete, the stack is reversed to obtain the topological ordering.



Java program for Topological sort using Queue (Kahn's Algorithm):

TopologicalSort_DFS.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

class TopologicalSort
 {
   // Topo sort only exists in DAGs i.e. Direct Acyclic Graph
   void dfs(List<Integer>[] adj, List<Integer> vis, int node, int n, Stack<Integer> stck)
    {
      vis.set(node, 1);
      for (int it : adj[node])
```

```
        {
            if (vis.get(it) == 0)
            {
                dfs(adj, vis, it, n, stck);
            }
        }
        stck.push(node);
    }

    // During the traversal u must be visited before v
    Stack<Integer> topo_sort(List<Integer>[] adj, int n)
    {
        List<Integer> vis = new ArrayList<>(n);
        for (int i = 0; i < n; i++)
        {
            vis.add(0);
        }
                // using stack ADT
        Stack<Integer> stck = new Stack<>();
        for (int i = 0; i < n; i++)
         {
            if (vis.get(i) == 0)
            {
                dfs(adj, vis, i, n, stck);
            }
        }
        return stck;
    }
}
public class TopologicalSort_DFS
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("enter number of vertices ");
        int n=sc.nextInt();

        System.out.println("enter number of  edges");
        int e=sc.nextInt();
        List<Integer>[] adj = new ArrayList[n];
        for (int i = 0; i < n; i++)
        {
            adj[i] = new ArrayList<>();
        }
```

```
        System.out.println("enter edges");
        for(int i=0;i<e;i++)
        {
            int end1=sc.nextInt();
            int end2=sc.nextInt();
            addEdge(adj,end1,end2);
        }
        TopologicalSort ts = new TopologicalSort();
        Stack<Integer> ans = ts.topo_sort(adj, n);
        while (!ans.isEmpty()) {
            int node = ans.pop();
            System.out.print(node + " ");
        }
    }
}
```

Example-1:

input=

```
enter number of vertices
6
enter number of  edges
9
enter edges
5 0
5 2
2 0
2 3
3 0
3 1
1 0
4 0
4 1
```

output=

```
topological order is
5 4 2 3 1 0
```

1. Parallel Courses:

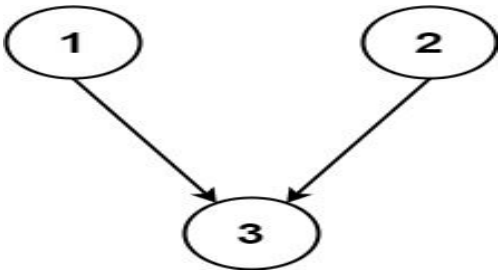
1.1. Parallel Courses-I:

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.

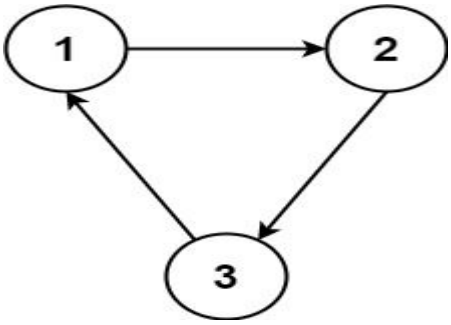
Return the minimum number of semesters needed to take all courses. If there is no way to take all the courses, return -1.

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

Java program for Implementing Parallel Course Application: ParallelCourses.java

```
import java.util.*;

public class Parallelcourses
{
   public int minimumSemesters(int N, int[][] relations)
   {
      // Step 1: Create adjacency list and indegree array
      List<List<Integer>> adj = new ArrayList<>();
      for (int i = 0; i <= N; i++)
       {
         adj.add(new ArrayList<>());
      }

      int[] indegree = new int[N + 1];
      for (int[] rel : relations)
      {
         int u = rel[0];
         int v = rel[1];
         adj.get(u).add(v);
         indegree[v]++;
      }
      // Step 2: Initialize queue with nodes having indegree 0
      Queue<Integer> queue = new LinkedList<>();
      for (int i = 1; i <= N; i++)
      {
         if (indegree[i] == 0)
         {
            queue.offer(i);
         }
```

```
    }

    int semester = 0;
    int completedCourses = 0;

    // Step 3: Topological sort with level/semester tracking
    while (!queue.isEmpty())
    {
        int size = queue.size(); // Number of courses you can take in current semester
        semester++;

        for (int i = 0; i < size; i++)
        {
            int course = queue.poll();
            completedCourses++;

            for (int neighbor : adj.get(course))
            {
                indegree[neighbor]--;
                if (indegree[neighbor] == 0)
                 {
                    queue.offer(neighbor);
                }
            }
        }
    }

    // Step 4: Check if all courses are completed
    return completedCourses == N ? semester : -1;
}

public static void main(String[] args)
{
    Scanner sc = new Scanner(System.in);
    Parallelcourses cs = new Parallelcourses();

    // Read number of courses
    System.out.print("Enter the number of courses: ");
    int N = sc.nextInt();

    // Read number of relations
    System.out.print("Enter the number of relations (prerequisites): ");
    int M = sc.nextInt();

    // Create an array to store relations
```

```
        int[][] relations = new int[M][2];

        // Read the relations
        System.out.println("Enter the relations (prerequisite pairs in the format 'X Y'):");
        for (int i = 0; i < M; i++)
         {
            relations[i][0] = sc.nextInt();  // course X
            relations[i][1] = sc.nextInt();  // course Y
        }

        // Call the method to compute minimum semesters
                System.out.println("cs.minimumSemesters(N, relations));

        sc.close();
    }
}
```

Example-1:

input=

```
Enter the number of courses: 3
Enter the number of relations (prerequisites): 2
Enter the relations (prerequisite pairs in the format 'X Y'):
1 3
2 3
```

output=

```
2
```

Example-2:

input=

```
Enter the number of courses: 3
Enter the number of relations (prerequisites): 3
Enter the relations (prerequisite pairs in the format 'X Y'):
1 2
2 3
3 1
```

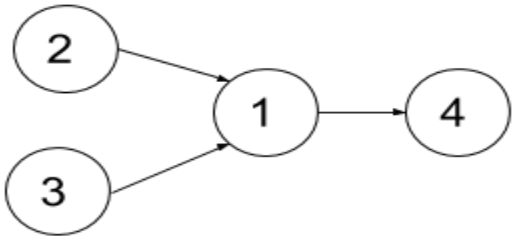
output=

```
-1
```

1.2.Parallel Courses-II:

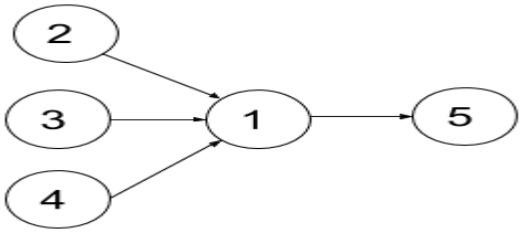
1. You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given an array relations where relations[i] = [prevCourse$_i$, nextCourse$_i$], representing a prerequisite relationship between course prevCourse$_i$ and course nextCourse$_i$: course prevCourse$_i$ has to be taken before course nextCourse$_i$. Also, you are given the integer k.
2. 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.
3. 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.

Java program for Implementing Parallel Course Application: ParallelCourses_II.java

```
import java.util.*;

public class Parallelcourses_II
 {

   public int minNumberOfSemesters(int n, int[][] relations, int k)
    {
      List<List<Integer>> graph = new ArrayList<>();
      int[] indegree = new int[n + 1];

      for (int i = 0; i <= n; i++)
      graph.add(new ArrayList<>());

      for (int[] rel : relations)
      {
         int u = rel[0], v = rel[1];
         graph.get(u).add(v);
         indegree[v]++;
      }
      Queue<Integer> queue = new LinkedList<>();
      for (int i = 1; i <= n; i++)
       {
         if (indegree[i] == 0)
         queue.offer(i);
      }
      int semesters = 0;
      while (!queue.isEmpty())
      {
         int size = queue.size();
         int canTake = Math.min(size, k);
         List<Integer> currentSemester = new ArrayList<>();
         for (int i = 0; i < canTake; i++)
         {
            currentSemester.add(queue.poll());
         }
         for (int course : currentSemester)
         {
            for (int neighbor : graph.get(course))
             {
               indegree[neighbor]--;
               if (indegree[neighbor] == 0)
```

```
                {
                    queue.offer(neighbor);
                }
            }
        }

        semesters++;
    }
    return semesters;
  }
  public static void main(String[] args)
 {
    Scanner sc = new Scanner(System.in);
    Parallelcourses_II courses = new Parallelcourses_II();
    // Read number of courses
    System.out.print("Enter number of courses (n): ");
    int n = sc.nextInt();
    // Read number of relations
    System.out.print("Enter number of prerequisite relations: ");
    int m = sc.nextInt();
    int[][] relations = new int[m][2];
    System.out.println("Enter prerequisite pairs (a b) :");

    for (int i = 0; i < m; i++) {
        relations[i][0] = sc.nextInt();
        relations[i][1] = sc.nextInt();
    }
    // Read k (max courses per semester)
    System.out.print("Enter maximum number of courses per semester (k): ");
    int k = sc.nextInt();

    int result = courses.minNumberOfSemesters(n, relations, k);
    System.out.println("Minimum number of semesters: " + result);
  }
}
```

Example-1:

input=

```
Enter number of courses (n): 4
Enter number of prerequisite relations: 3
Enter prerequisite pairs (a b) :
2 1
3 1
1 4
Enter maximum number of courses per semester (k): 2
```

output=

```
Minimum number of semesters: 3
```

2. Course Schedule Application:

2.1. Course Schedule-I:

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a_i, b_i] indicates that you must take course bi first if you want to take course a_i.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return true if you can finish all courses. Otherwise, return false.

Example 1:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible.

Example 2:
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Java Program for Implementing Course Schedule-I Application: CourseSchedule_I.java

```
import java.util.*;

public class CourseSchedule_I
 {

   public boolean canFinish(int numCourses, int[][] prerequisites)
   {
      // Step 1: Build graph and indegree array
      List<List<Integer>> graph = new ArrayList<>();
      int[] indegree = new int[numCourses];

      for (int i = 0; i < numCourses; i++)
         graph.add(new ArrayList<>());

      for (int[] pre : prerequisites) {
         int course = pre[0];
         int prereq = pre[1];
         graph.get(prereq).add(course); // prereq -> course
```

```
            indegree[course]++;
        }

        // Step 2: Initialize queue with courses having 0 prerequisites
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0)
                queue.offer(i);
        }

        int finishedCourses = 0;

        // Step 3: Process courses
        while (!queue.isEmpty()) {
            int current = queue.poll();
            finishedCourses++;

            for (int neighbor : graph.get(current)) {
                indegree[neighbor]--;
                if (indegree[neighbor] == 0)
                    queue.offer(neighbor);
            }
        }

        // Step 4: If we finished all courses, return true
        return finishedCourses == numCourses;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        CourseSchedule_I cs = new CourseSchedule_I();

        System.out.print("Enter number of courses: ");
        int numCourses = sc.nextInt();

        System.out.print("Enter number of prerequisite pairs: ");
        int m = sc.nextInt();

        int[][] prerequisites = new int[m][2];
        System.out.println("Enter prerequisites");
        for (int i = 0; i < m; i++) {
            prerequisites[i][0] = sc.nextInt();
            prerequisites[i][1] = sc.nextInt();
        }
```

```
        boolean canFinish = cs.canFinish(numCourses, prerequisites);
        System.out.println(canFinish);
    }
}
```

Example-1:
input=
Enter number of courses: 2
Enter number of prerequisite pairs: 1
Enter prerequisites:
1 0
output=
true

Example-2:
input=
Enter number of courses: 2
Enter number of prerequisite pairs: 2
Enter prerequisites:
1 0
0 1
output=
false

2.2. Course Schedule-II:

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1.
You are given an array prerequisites where prerequisites[i] = [a_i, b_i]
indicates that you must take course bi first if you want to take course a_i.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return the ordering of courses you should take to finish all courses.
If there are many valid answers, return any of them.
If it is impossible to finish all courses, return an empty array.

Example 1:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

Example 2:
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.
So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

Example 3:
Input: numCourses = 1, prerequisites = []
Output: [0]

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.

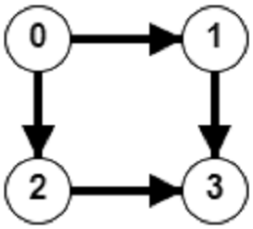
Approach:

- 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.

Java Program for Implementing Course Schedule-II Application: CourseSchedule-II.java

```
import java.util.*;

class CourseSchedule_II

{
  public int[] findOrder(int numCourses, int[][] prerequisites)
   {
     // first task is to create graph..
     List<List<Integer>> graph = new ArrayList<>();
     for(int i = 0 ; i < numCourses; i++)
     {
        graph.add(new ArrayList<>());
     }

     for(int i = 0 ; i < prerequisites.length; i++)
     {
        graph.get(prerequisites[i][1]).add(prerequisites[i][0]);
```

```
    }

    // create an indegree array for each node...
    int[] indegree = new int[numCourses];

    for(int i = 0; i < numCourses ; i++)
    {
        for(int node : graph.get(i))
        {
            indegree[node]++;
        }
    }

    // now adding node to q, which have indegree 0...
    Queue<Integer> q = new LinkedList<>();
    for(int i = 0; i < indegree.length; i++)
    {
        if(indegree[i] == 0){
            q.add(i);
        }
    }


    // topological sorted array..
    int[] ts = new int[numCourses];
    int i = 0;

    while(!q.isEmpty())
    {
        int node = q.remove();
        ts[i++] = node;

        for(int nbr : graph.get(node))
        {
            indegree[nbr]--;
            if(indegree[nbr] == 0)
            {
                q.add(nbr);
            }
        }

    }
    if(i == 0 || i < numCourses) return new int[]{};
    return ts;
}
```

```
   public static void main(String[] args)
    {
      Scanner sc = new Scanner(System.in);
      CourseSchedule_II  cs = new CourseSchedule_II ();

      int numCourses = sc.nextInt();
      int m = sc.nextInt();
      int[][] prerequisites = new int[m][2];
     for (int i = 0; i < m; i++)
       {
        prerequisites[i][0] = sc.nextInt();
        prerequisites[i][1] = sc.nextInt();
      }

      System.out.println( cs.findOrder(numCourses, prerequisites));

   }
}
```

Example-1:
input=
2
1
1 0
output=
[0,1]

Example-2:
input=
4
4
1 0
2 0
3 1
3 2
output=
[0,2,1,3]

Example-3:
input=
1
0
output=
[ ]