Classroom Theory

− Introduction to Graphs

A *Graph* is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

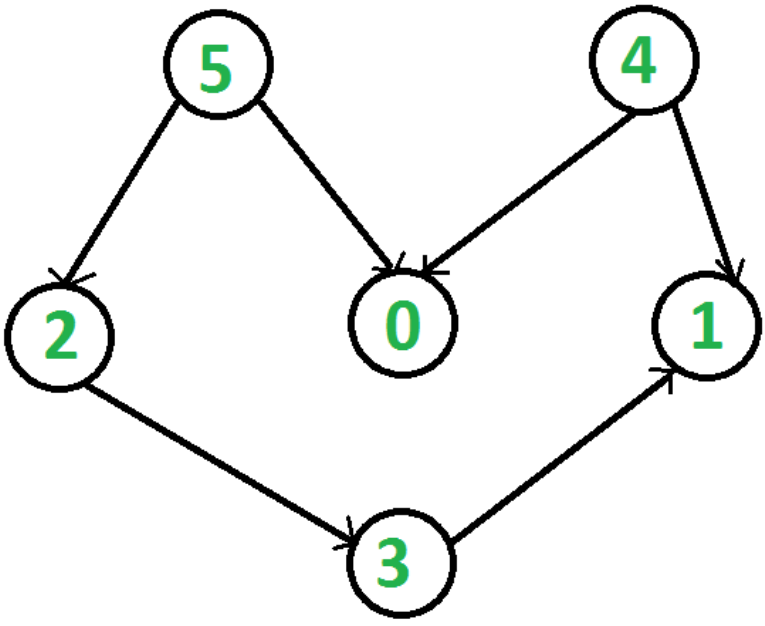
Graphs are used to represent many real-life applications:

- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example Google GPS
- Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Directed and Undirected Graphs

- **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction.

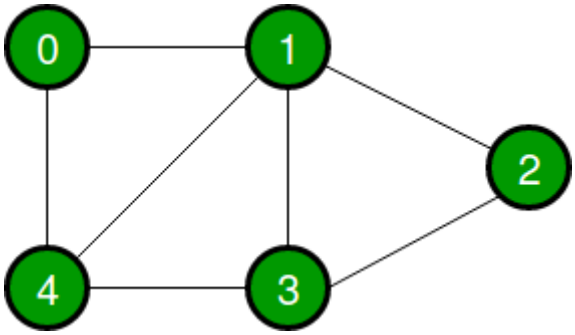
  For Example, the below graph is a directed graph:



- **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between

vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u**.

Following is an example of an undirected graph with 5 vertices:



Representing Graphs

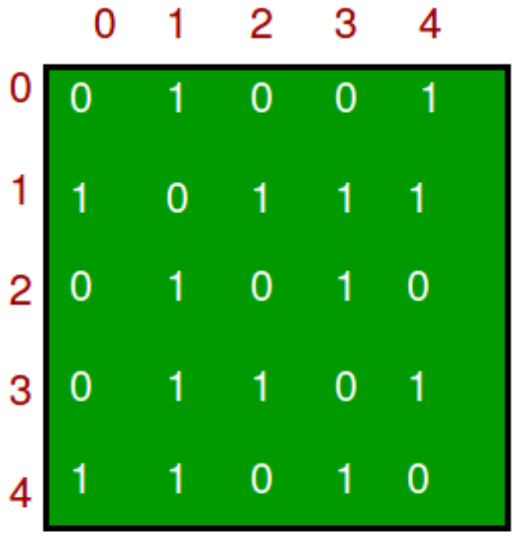
Following two are the most commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency List.

Let us look at each one of the above two method in details:

- **Adjacency Matrix:** The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

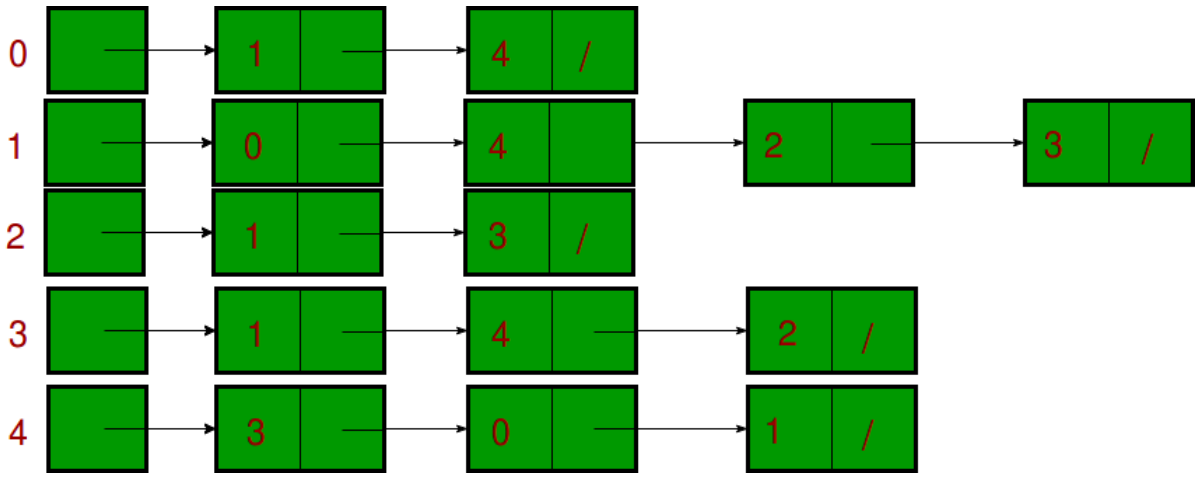
  The adjacency matrix for the above example undirected graph is:



Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time. Please see this (https://ide.geeksforgeeks.org/9je5j6jJ13) for a sample Python implementation of adjacency matrix.

- **Adjacency List:** Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



Below is the implementation of the adjacency list representation of Graphs:

Note: In below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of a linked list. The vector implementation has advantages of cache friendliness.

C++

```
// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;
// A utility function to add an edge in an undirected grap
void addEdge(vector<int> adj[], int u, int v){ adj[u].push
// A utility function to print the adjacency list represer
void printGraph(vector<int> adj[], int V){
    for (int v = 0; v < V; ++v){
        cout << "\n Adjacency list of vertex "<< v << "\n
        for (auto x : adj[v]) cout << "-> " << x;
        printf("\n");
    }
}
int main(){
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    printGraph(adj, V);
    return 0;
}

```

Run

Java

C++

Java

```java
// Java code to demonstrate Graph representation using Arr
import java.util.*;
class Graph {
    // A utility function to add an edge in an undirected
    static void addEdge(ArrayList<ArrayList<Integer> > adj
        adj.get(u).add(v);adj.get(v).add(u);
    }
    // A utility function to print the adjacency list repr
    static void printGraph(ArrayList<ArrayList<Integer> >
        for (int i = 0; i < adj.size(); i++) {
            System.out.println("\nAdjacency list of verte
            for (int j = 0; j < adj.get(i).size(); j++) {
                System.out.print(" -> "+adj.get(i).get(j))
            }
            System.out.println();
        }
    }
    public static void main(String[] args){
        // Creating a graph with 5 vertices
        int V = 5;
        ArrayList<ArrayList<Integer> > adj = new ArrayList

        for (int i = 0; i < V; i++) adj.add(new ArrayList<

        // Adding edges one by one
        addEdge(adj, 0, 1); addEdge(adj, 0, 4); addEdge(ad
        addEdge(adj, 1, 4); addEdge(adj, 2, 3); addEdge(ad
        printGraph(adj);
    }
}
```

Run

Output:

```
Adjacency list of vertex 0
head -> 1-> 4

Adjacency list of vertex 1
head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2
head -> 1-> 3

Adjacency list of vertex 3
head -> 1-> 2-> 4

Adjacency list of vertex 4
head -> 0-> 1-> 3
```

Pros: Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

− Breadth First Traversal of a Graph

The ***Breadth First Traversal*** or ***BFS*** traversal of a graph is similar to that of the Level Order Traversal of Trees.

The BFS traversal of Graphs also traverses the graph in levels. It starts the traversal with a given vertex, visits all of the vertices adjacent to the initially given vertex and pushes them all to a queue in order of visiting. Then it pops an element from the front of the queue, visits all of its neighbours and pushes the neighbours which are not already visited into the queue and repeats the process until the queue is empty or all of the vertices are visited.

The BFS traversal uses an auxiliary boolean array say *visited[]* which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.

Complete Algorithm:

1. Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
2. Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.

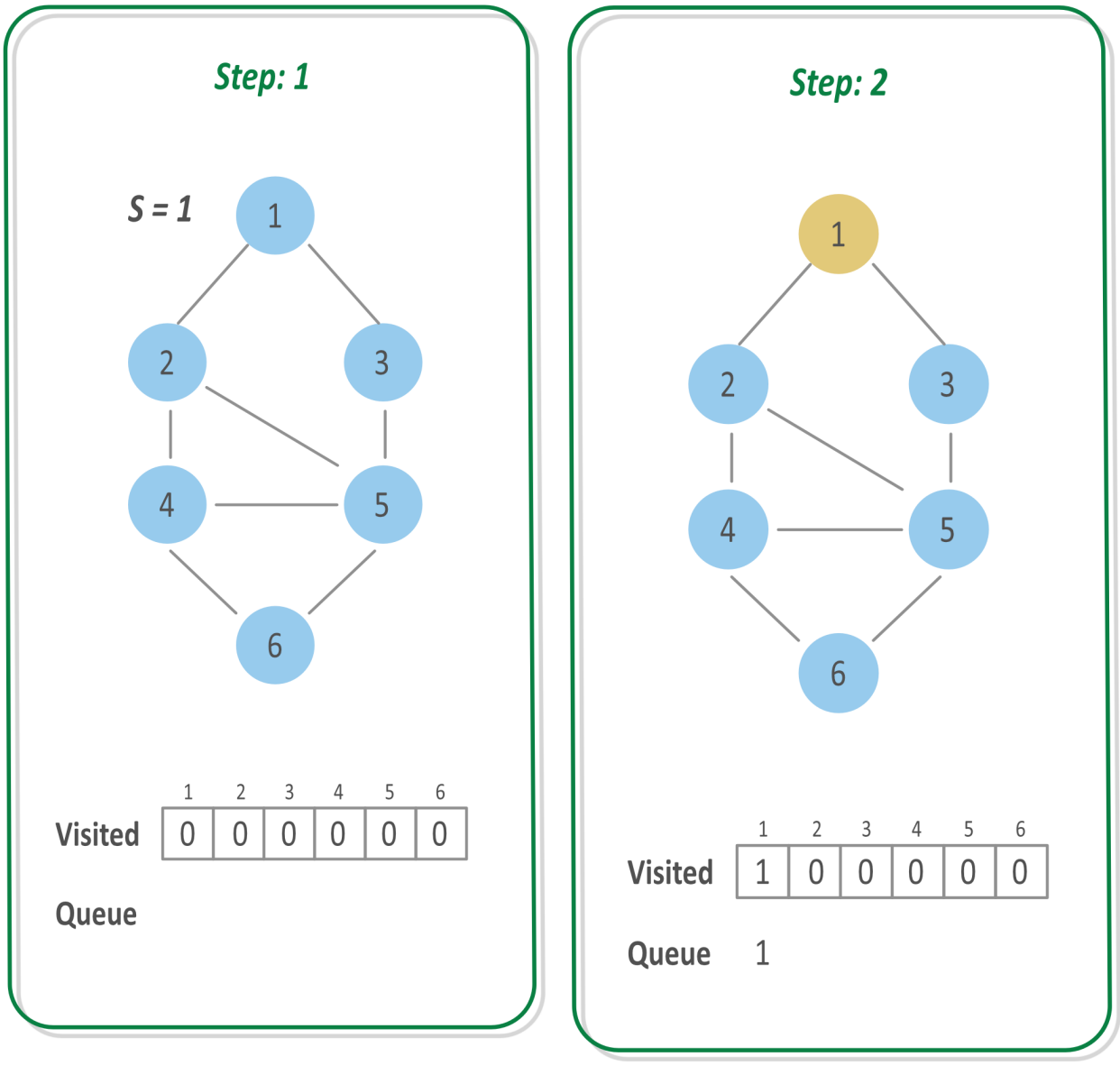
3. Until the Queue is non-empty, repeat the below steps:
    - Pop an element from the queue and print the popped element.
    - Traverse all of the vertices adjacent to the vertex poped from the queue.
    - If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.

Illustration:

Consider the graph shown in the below Image. The vertices marked **blue** are *not-visited* vertices and the vertices marked **yellow** are *visited*. The vertex numbered **1** is the source vertex, i.e. the BFS traversal will start from the vertex 1.

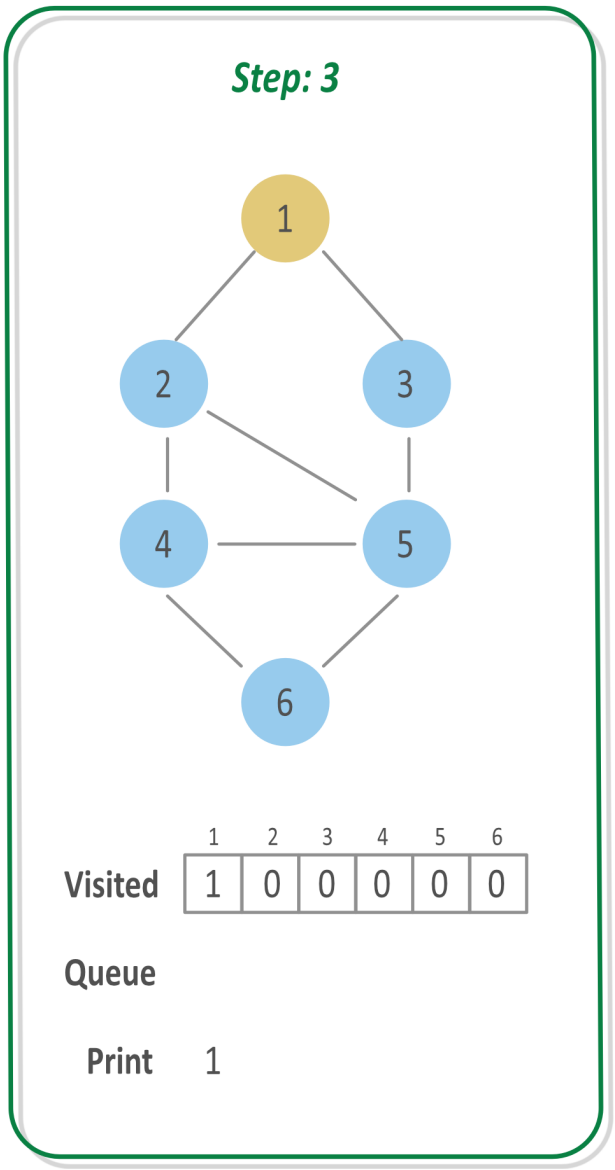
Following the BFS algorithm:

- Mark the vertex 1 visited in the visited[] array and push it to the queue.

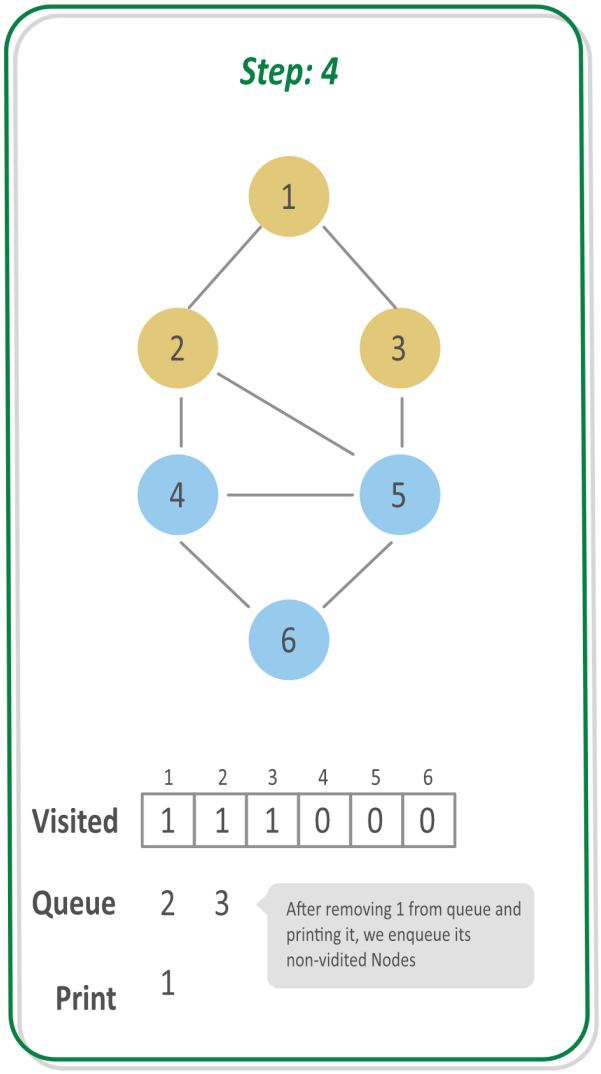


Since the Queue is empty now, it means that the complete graph is traversed.

Step 3: POP the vertex at the front of queue that is 1, and print it.



Step 4: Check if adjacent vertices of the vertex 1 are not already visited. If not, mark them visited and push them back to the queue.

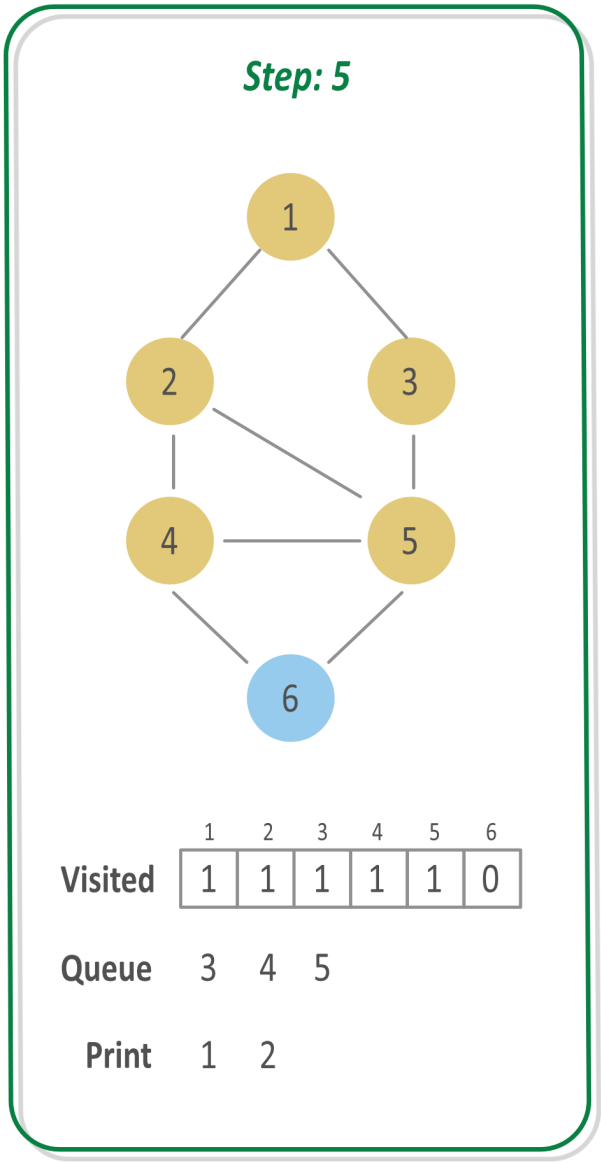


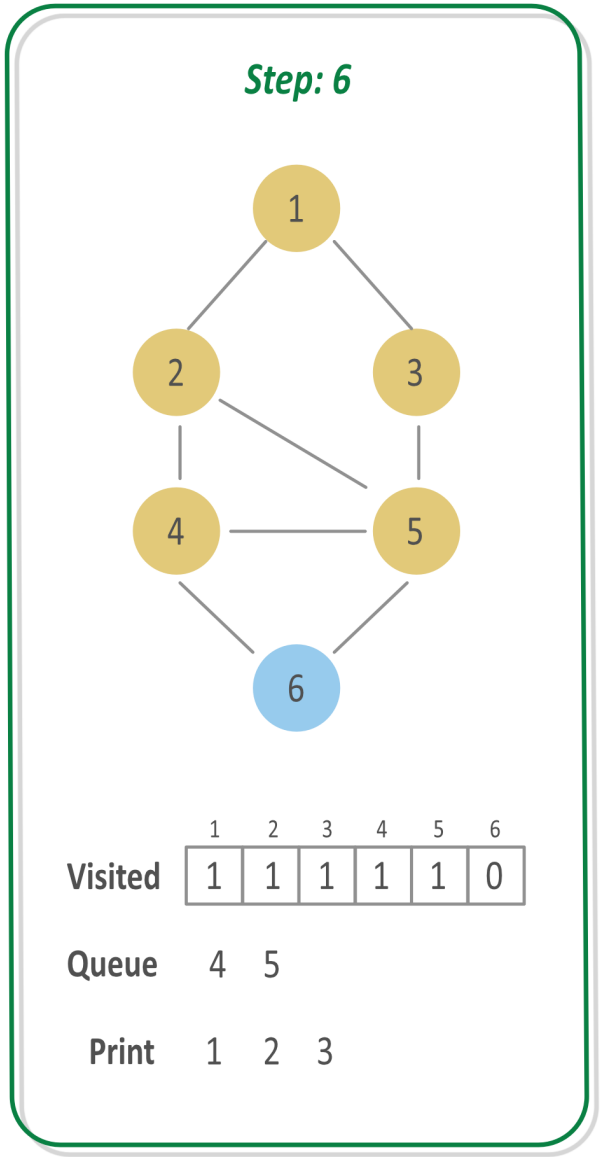
Step 5:

- POP the vertex at front that is 2 and print it.
- Check if the adjacent vertices of 2 are not already visited. If not, mark them visited and push them to queue. So, push 4 and 5.

Step 6:

- POP the vertex at front that is 3 and print it.
- Check if the adjacent vertices of 3 are not already visited. If not, mark them visited and push them to queue. So, donot push anything.



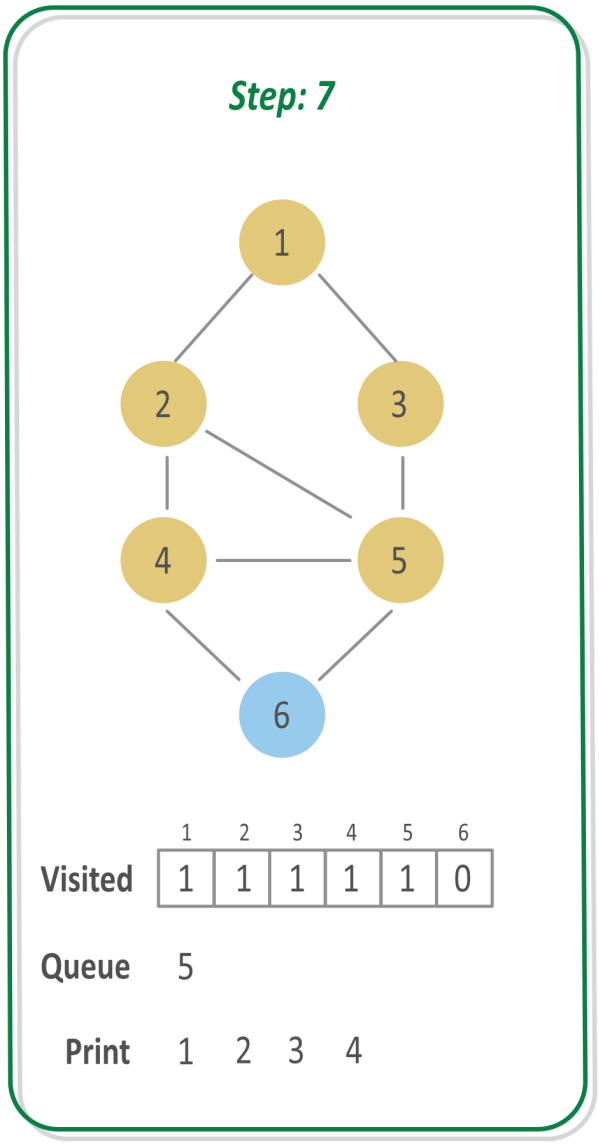


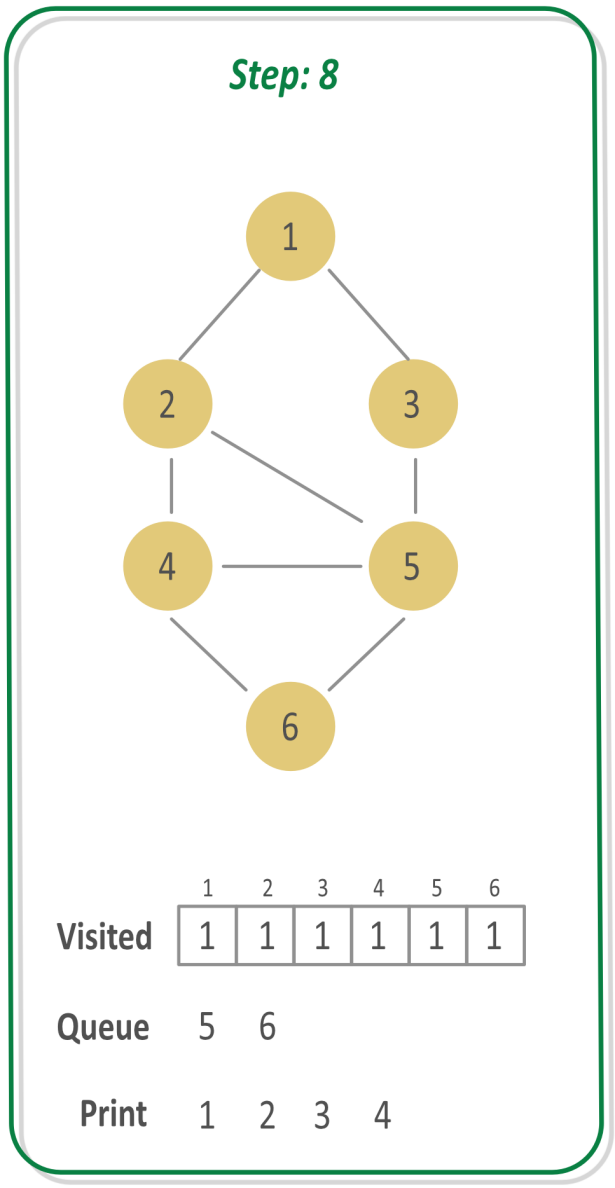
Step 7:

- POP the vertex at front that is 4 and print it.

Step 8:

- Check if the adjacent vertices of 4 are not already visited. If not, mark them visited and push them to queue. So, push 6 to the queue.



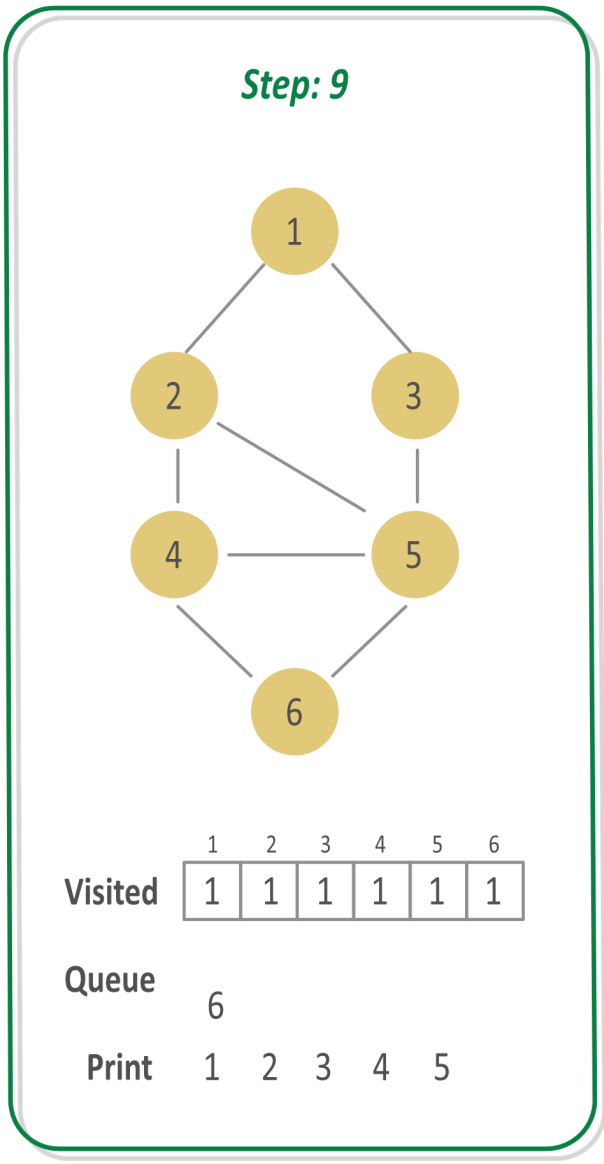


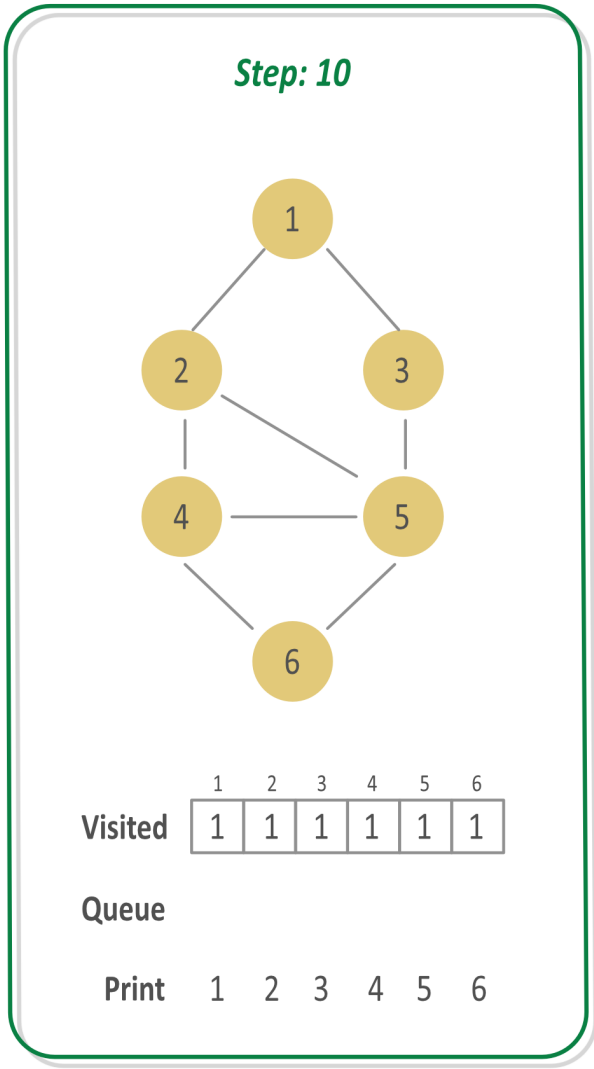
Step 9:

- POP the vertex at front, that is 5 and print it.
- Since, all of its adjacent vertices are already visited, donot push anything.

Step 10:

- POP the vertex at front, that is 6 and print it.
- Since, all of its adjacent vertices are already visited, donot push anything.





Implementation:

C++

```
// C++ program to implement BFS traversal
// of a Graph

#include <bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Function to perform BFS traversal of the given Graph
void BFS(vector<int> adj[], int V)
{
    // Initialize a boolean array
    // to keep track of visited vertices
```

```
21      bool visited[V + 1];
22
23      // Mark all vertices not-visited initially
24      for (int i = 1; i <= V; i++)
25          visited[i] = false;
26
27      // Create a Queue to perform BFS
28      queue<int> q;
29
30      // Our source vertex is vertex
```

Run

Java

Complete Code/Output:

```
// C++ program to implement BFS traversal
// of a Graph

#include <bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Function to perform BFS traversal of the given Graph
void BFS(vector<int> adj[], int V)
{
    // Initialize a boolean array
    // to keep track of visited vertices
    bool visited[V + 1];

    // Mark all vertices not-visited initially
    for (int i = 1; i <= V; i++)
        visited[i] = false;

    // Create a Queue to perform BFS
    queue<int> q;

    // Our source vertex is vertex
    // numbered 1
    int s = 1;

    // Mark S visited and Push to queue
    visited[s] = true;
    q.push(s);

    while (!q.empty()) {
        // Pop element at front and print
        int node = q.front();
        q.pop();

        cout << node << " ";

        // Traverse the nodes adjacent to the currently
        // poped element and push those elements to the
        // queue which are not already visited
        for (int i = 0; i < adj[node].size(); i++) {
            if (visited[adj[node][i]] == false) {
```

```
                // Mark it visited
                visited[adj[node][i]] = true;

                // Push it to the Queue
                q.push(adj[node][i]);
            }
        }
    }
}

// Driver code
int main()
{
    int V = 6;
    vector<int> adj[V + 1];
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 2, 4);
    addEdge(adj, 2, 5);
    addEdge(adj, 3, 5);
    addEdge(adj, 4, 5);
    addEdge(adj, 4, 6);
    addEdge(adj, 5, 6);

    BFS(adj, V);

    return 0;
}
```

```
// Java code to illustrate BFS traversal
// in a Graph

import java.util.*;

class Graph {

    // A utility function to add an edge in an
    // undirected graph
    static void addEdge(ArrayList<ArrayList<Integer> > adj,
                        int u, int v)
    {
        adj.get(u).add(v);
        adj.get(v).add(u);
    }

    // Function to perform BFS traversal of a Graph
    static void BFS(ArrayList<ArrayList<Integer> > adj, int V)
    {
        // Initialize a boolean array
        // to keep track of visited vertices
        boolean visited[] =  new boolean[V+1];

        // Mark all vertices not-visited initially
        for (int i = 1; i <= V; i++)
            visited[i] = false;

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // The start vertex or source vertex is 1
        int s = 1;

        // Mark the current node as
        // visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Traverse the nodes adjacent to the currently
            // poped element and push those elements to the
            // queue which are not already visited
            for (int i = 0; i < adj.get(s).size(); i++) {
```

```
                // Fetch adjacent node
                int newNode = adj.get(s).get(i);

                // Check if it is not visited
                if(visited[newNode] == false)
                {
                    // Mark it visited
                    visited[newNode] = true;

                    // Add it to queue
                    queue.add(newNode);
                }
            }
        }
    }

    // Driver Code
    public static void main(String[] args)
    {
        // Creating a graph with 6 vertices
        int V = 6;
        ArrayList<ArrayList<Integer> > adj
                    = new ArrayList<ArrayList<Integer> >(V+1);

        for (int i = 0; i < V+1; i++)
            adj.add(new ArrayList<Integer>());

        // Adding edges one by one
        addEdge(adj, 1, 2);
        addEdge(adj, 1, 3);
        addEdge(adj, 2, 4);
        addEdge(adj, 2, 5);
        addEdge(adj, 3, 5);
        addEdge(adj, 4, 5);
        addEdge(adj, 4, 6);
        addEdge(adj, 5, 6);

        BFS(adj, V);
    }
}
```

```
1 2 3 4 5 6
```

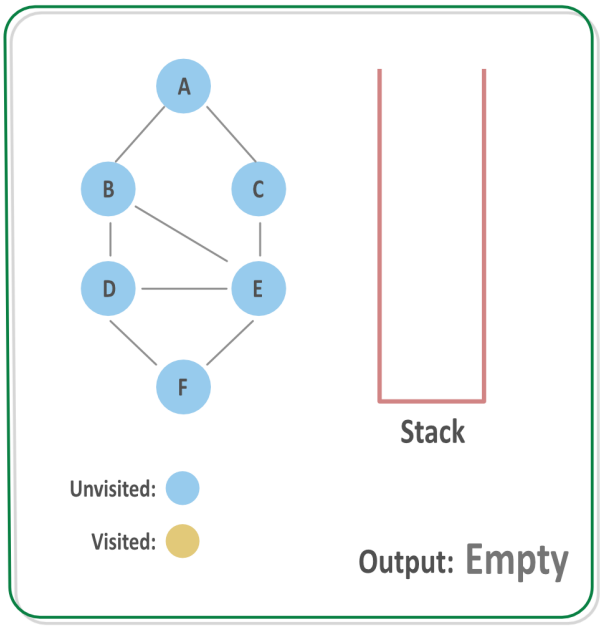
- Depth First Traversal of a Graph

The Depth-First Traversal or the DFS traversal of a Graph is used to traverse a graph depth wise. That is, it in this traversal method, we start traversing the graph from a node and keep on going in the same direction as far as possible. When no nodes are left to be traversed along the current path, backtrack to find a new possible path and repeat this process until all of the nodes are visited.

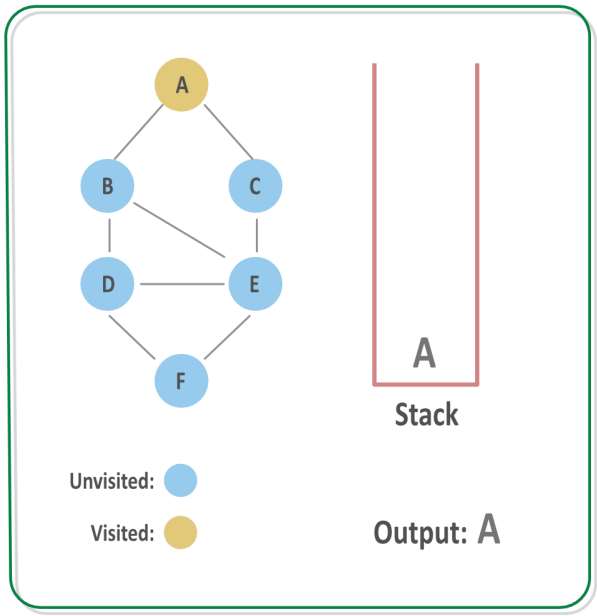
We can implement the DFS traversal algorithm using a recursive approach. While performing the DFS traversal the graph may contain a cycle and the same node can be visited again, so in order to avoid this we can keep track of visited array using an auxiliary array. On each step of the recursion mark, the current vertex visited and call the recursive function again for all the adjacent vertices.

Illustration:

Step 1: Consider the below graph and apply the DFS algorithm recursively for every node using an auxiliary stack for recursive calls and an array to keep track of visited vertices.



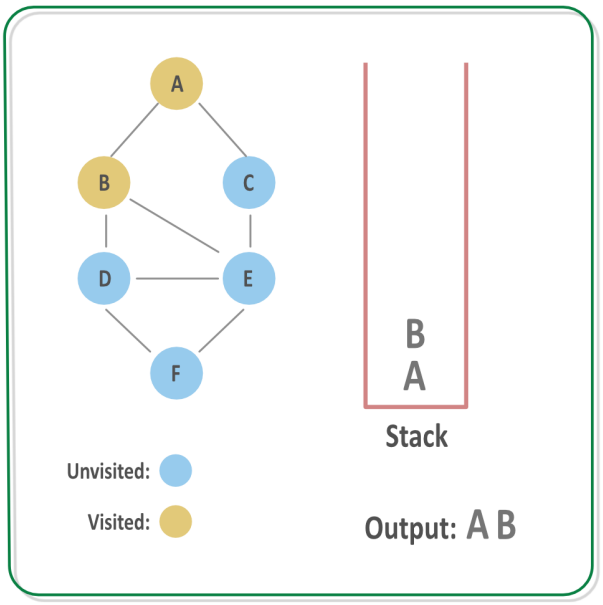
Step 2: Process the vertex A, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is B. The vertex A is put on the auxilary stack for now.

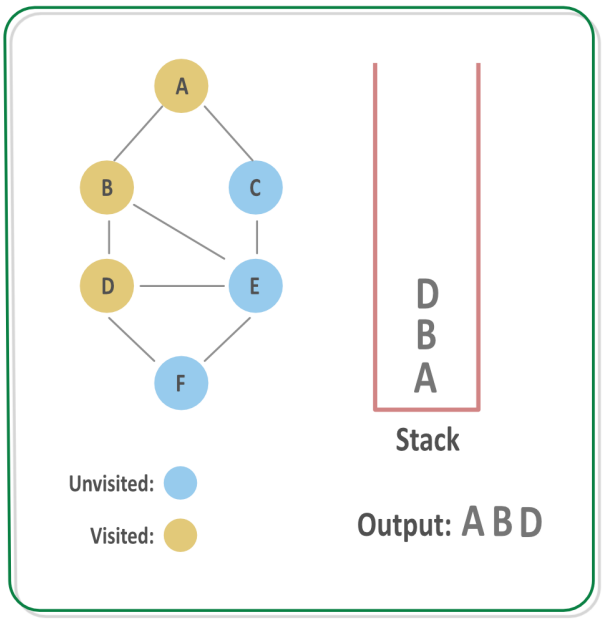


Step 3: Process the vertex B, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is D. The vertex B is put on the auxilary stack for now.

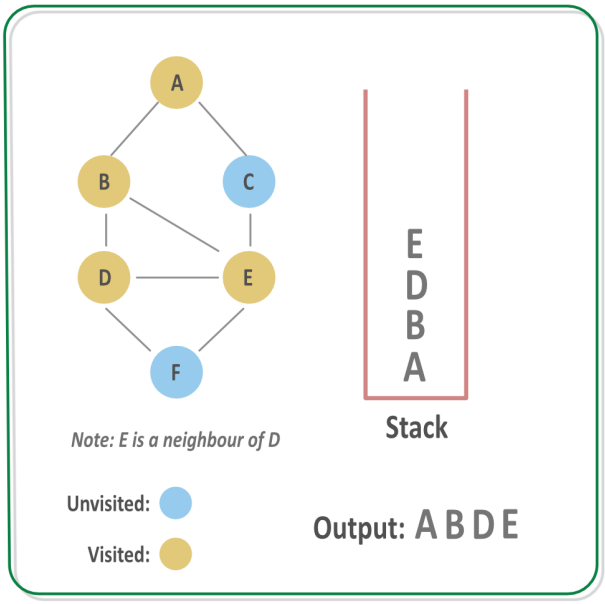
Step 4: Process the vertex D, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is E. The vertex D is put on the auxilary stack for now.

Step 9: Since there are no adjacent vertices

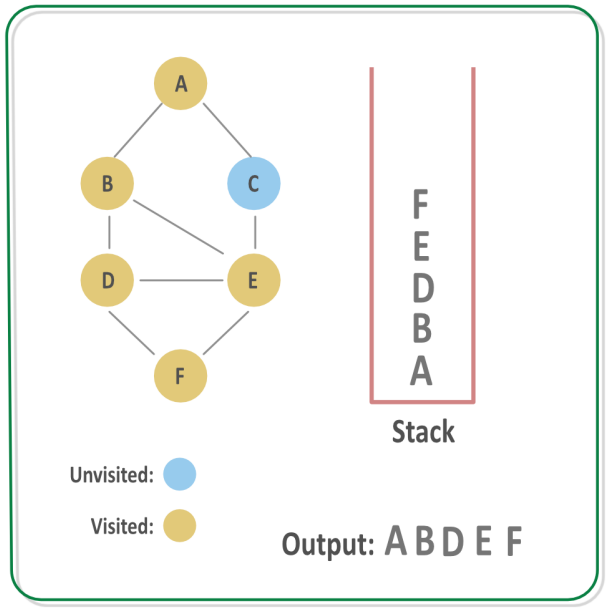




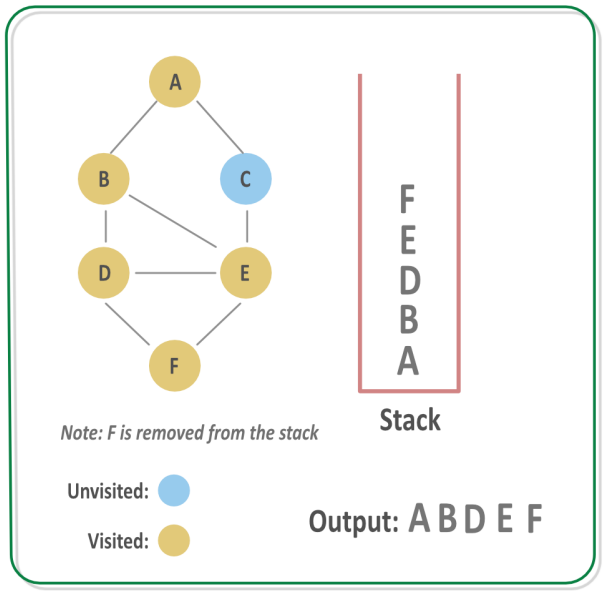
Step 5: Process the vertex E, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is F. The vertex E is put on the auxilary stack for now.



Step 6: Process the vertex F, mark it visited and call DFS for its adjacent vertices. There are no adjacent vertex of the vertex F, so backtrack to find a new path. The vertex F is put on the auxilary stack for now.

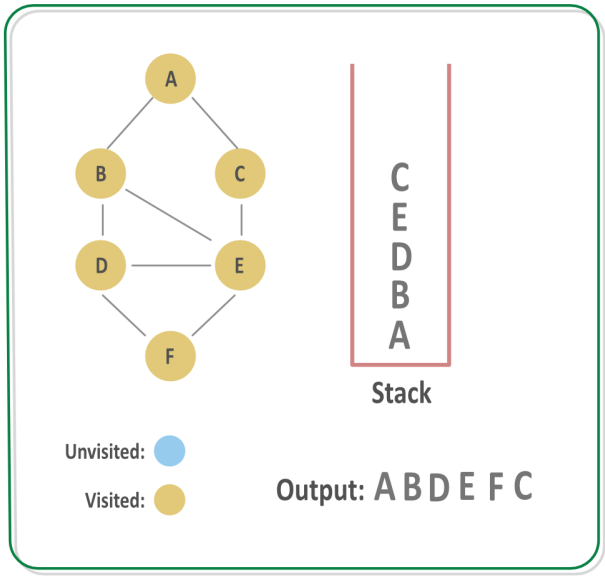


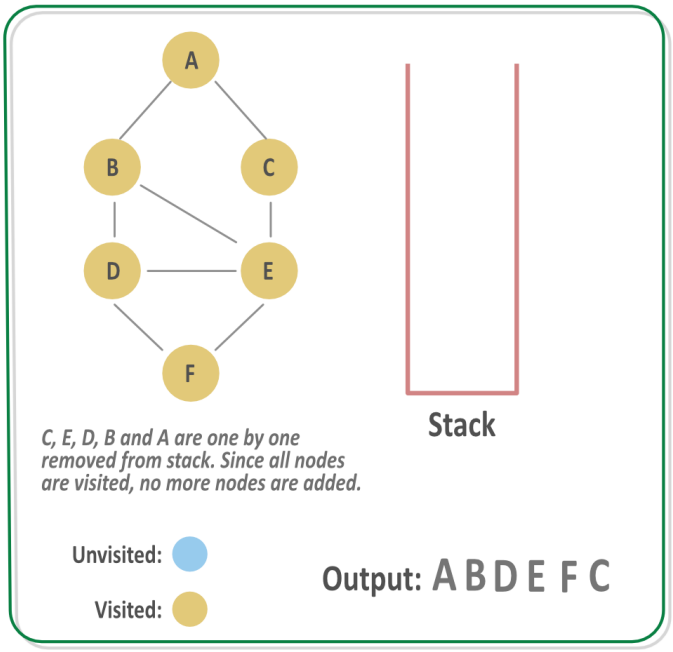
Step 7: Since the vertex F has no adjacent vertices left unvisited, backtrack and go to previous call, that is process any other adjacent vertex of node E, that is C.



w path and keep removing nodes from stack until a new path is found. Since all of the nodes are processed so the stack will get empty.

Step 8: Process the vertex C, mark it visited and call DFS for its adjacent vertices. The vertex C is put on the auxilary stack for now.





Implementation:

C++

```
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

```

Run

Java

Complete Code/Output:

```
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
```

```
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

```
// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices

    // Array  of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(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.
    }

    // A function used by DFS
    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
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
```

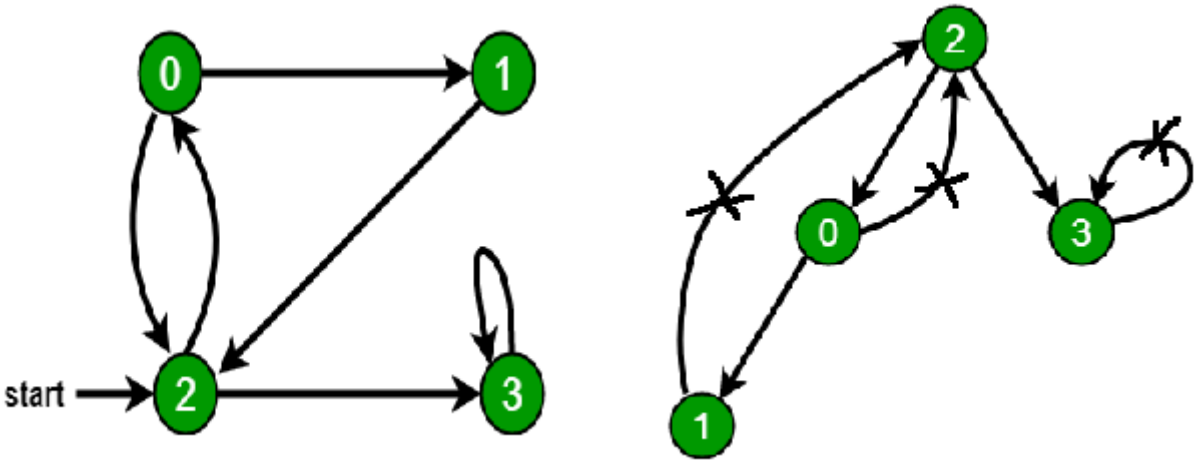
```
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal "+
                           "(starting from vertex 2)");

        g.DFS(2);
    }
}
```

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

- Detecting Cycle in a Graph

Problem: Given a graph(directed or undirected), check whether the graph contains a cycle or not.

Solution: Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge (http://en.wikipedia.org/wiki/Depth-first_search#Output_of_a_depth-first_search) present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



(https://media.geeksforgeeks.org/wp-content/uploads/cycle.png)
To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. We can use an auxiliary array say, *recStack[]* to keep track of vertices in the recursion stack.

Therefore, for every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not a parent of v, then there is a cycle in the graph. If we don't find such an adjacent for any vertex, we say that there is no cycle.

Note: The above method can be used to detect a cycle in both *directed* and *undirected graphs*.

Below is the implementation of the above aprroach:

C++

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adj
    bool isCyclicUtil(int v, bool visited[], bool *rs);  //
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle
};

Graph::Graph(int V)
{
```

```
22      this->V = V;
23      adj = new list<int>[V];
24  }
25
26  void Graph::addEdge(int v, int w)
27  {
28      adj[v].push_back(w); // Add w to v's list.
29  }
30
```

Run

Java

Complete Code/Output:

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], bool *rs);  // used by isCyclic
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

// Utility function to detect cycle in a Graph
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }
    }
```

```
    recStack[v] = false;  // remove the vertex from recursion stack
    return false;
}

// Returns true if the graph contains a cycle, else false
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and
    // not part of recursion stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

    // Call the recursive helper function to detect
    // cycle in different DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

// Driver Code
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if(g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";
    return 0;
}
```

```
// A Java Program to detect cycle in a graph
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

class Graph {

    private final int V;
    private final List<List<Integer>> adj;

    public Graph(int V)
    {
        this.V = V;
        adj = new ArrayList<>(V);

        for (int i = 0; i < V; i++)
            adj.add(new LinkedList<>());
    }

    // Utility Function to check cycle in a Graph
    private boolean isCyclicUtil(int i, boolean[] visited,
                                       boolean[] recStack)
    {

        // Mark the current node as visited and
        // part of recursion stack
        if (recStack[i])
            return true;

        if (visited[i])
            return false;

        visited[i] = true;

        recStack[i] = true;
        List<Integer> children = adj.get(i);

        for (Integer c: children)
            if (isCyclicUtil(c, visited, recStack))
                return true;

        recStack[i] = false;

        return false;
    }

    private void addEdge(int source, int dest) {
        adj.get(source).add(dest);
```

```
    }

    // Returns true if the graph contains a
    // cycle, else false.
    // This function is a variation of DFS() in
    // https://www.cdn.geeksforgeeks.org/archives/18212
    private boolean isCyclic()
    {
        // Mark all the vertices as not visited and
        // not part of recursion stack
        boolean[] visited = new boolean[V];
        boolean[] recStack = new boolean[V];


        // Call the recursive helper function to
        // detect cycle in different DFS trees
        for (int i = 0; i < V; i++)
            if (isCyclicUtil(i, visited, recStack))
                return true;

        return false;
    }

    // Driver code
    public static void main(String[] args)
    {
        Graph graph = new Graph(4);
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);
        graph.addEdge(2, 3);
        graph.addEdge(3, 3);

        if(graph.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't "
                                    + "contain cycle");
    }
}
```

```
Graph contains cycle
```

The *Time Complexity* of this method is same as the time complexity of *DFS traversal*

which is O(V+E), where V is the number of vertices and E is the number of edges.

- Dijkstra's Algorithm for Shortest Path in a Weighted Graph

> *Given a graph and a source vertex in the graph, find the shortest paths from source to all vertices in the given graph.*

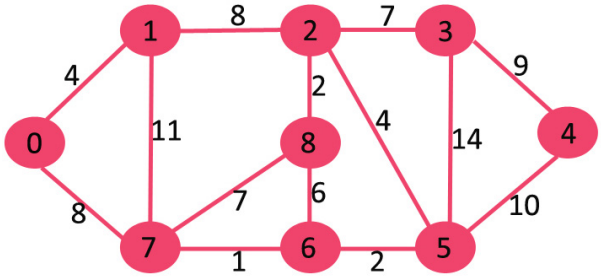
Dijkstra's algorithm is a variation of the BFS algorithm. In Dijkstra's Algorithm, a SPT *(shortest path tree)* is generated with given source as root. Each node at this SPT stores the value of the shortest path from the source vertex to the current vertex. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below is the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given weighted graph.

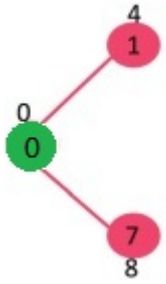
Algorithm:

1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While *sptSet* doesn't include all vertices:
    - Pick a vertex u which is not there in *sptSet* and has minimum distance value.
    - Include u to *sptSet*.
    - Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

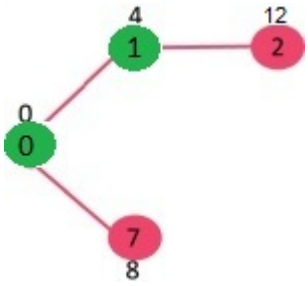
Let us understand the above algorithm with the help of an example. Consider the below given graph:



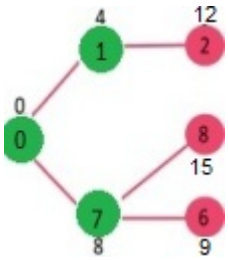
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



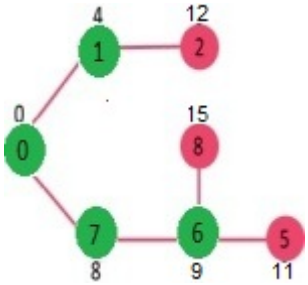
(https://media.geeksforgeeks.org/wp-content/cdn-uploads/MST1.jpg)
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/DIJ2.jpg)
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/DIJ3.jpg)
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/DIJ4.jpg)
We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

Implementation:
Since at every step we need to find the vertex with minimum distance from the source vertex from the set of vertices currently not added to the SPT, so we can use a min heap for easier and efficient implementation. Below is the complete algorithm using priority_queue(min heap) to implement Dijkstra's Algorithm:

```
1) Initialize distances of all vertices as infinite.

2) Create an empty priority_queue pq. Every item
   of pq is a pair (weight, vertex). Weight (or
   distance) is used as the first item of pair
   as the first item is by default used to compare
   two pairs

3) Insert source vertex into pq and make its
   distance as 0.

4) While either pq doesn't become empty
    a) Extract minimum distance vertex from pq.
       Let the extracted vertex be u.
    b) Loop through all adjacent of u and do
       following for every vertex v.

           // If there is a shorter path to v
           // through u.
           If dist[v] > dist[u] + weight(u, v)

               (i) Update distance of v, i.e., do
                     dist[v] = dist[u] + weight(u, v)
               (ii) Insert v into the pq (Even if v is
                    already there)

5) Print distance array dist[] to print all shortest
   paths.
```

C++

```
 1
 2  // Program to find Dijkstra's shortest path using
 3  // min heap in STL
 4
 5  #include<bits/stdc++.h>
 6  using namespace std;
 7
 8  # define INF 0x3f3f3f3f
 9
10  // iPair ==> Integer Pair
11  typedef pair<int, int> iPair;
12
13  // To add an edge
14  void addEdge(vector <pair<int, int> > adj[], int u,
15                                          int v, int wt)
16  {
17      adj[u].push_back(make_pair(v, wt));
18      adj[v].push_back(make_pair(u, wt));
```

```
}

// Prints distance of shortest paths from the source
// vertex to all other vertices
void shortestPath(vector<pair<int,int> > adj[], int V, int s
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> >

```

Run

Java

Complete Code/Output:

```
// Program to find Dijkstra's shortest path using
// min heap in STL

#include<bits/stdc++.h>
using namespace std;

# define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
                                     int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

// Prints distance of shortest paths from the source
// vertex to all other vertices
void shortestPath(vector<pair<int,int> > adj[], int V, int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    // Create a vector for distances and initialize all
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in priority queue and initialize
    // its distance as 0.
    pq.push(make_pair(0, src));
    dist[src] = 0;

    /* Looping till priority queue becomes empty (or all
    distances are not finalized) */
    while (!pq.empty())
    {
        // The first vertex in pair is the minimum distance
        // vertex, extract it from priority queue.
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted distance (distance must be first item
        // in pair)
```

```
        int u = pq.top().second;
        pq.pop();

        // Get all adjacent of u.
        for (auto x : adj[u])
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = x.first;
            int weight = x.second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print shortest distances stored in dist[]
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Driver Code
int main()
{
    int V = 9;
    vector<iPair > adj[V];

    // making above shown graph
    addEdge(adj, 0, 1, 4);
    addEdge(adj, 0, 7, 8);
    addEdge(adj, 1, 2, 8);
    addEdge(adj, 1, 7, 11);
    addEdge(adj, 2, 3, 7);
    addEdge(adj, 2, 8, 2);
    addEdge(adj, 2, 5, 4);
    addEdge(adj, 3, 4, 9);
    addEdge(adj, 3, 5, 14);
    addEdge(adj, 4, 5, 10);
    addEdge(adj, 5, 6, 2);
    addEdge(adj, 6, 7, 1);
    addEdge(adj, 6, 8, 6);
    addEdge(adj, 7, 8, 7);
```

```
    shortestPath(adj, V, 0);

    return 0;
}
```

```
// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath {
    // A utility function to find the vertex with minimum distance value,
    // from the set of vertices not yet included in shortest path tree
    static final int V = 9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min) {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex   Distance from Source\n");
        for (int i = 0; i < V; i++)
            System.out.println(i + "                " + dist[i]+"\n");
    }

    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation
    void dijkstra(int graph[][], int src)
    {
        int dist[] = new int[V]; // The output array. dist[i] will hold
        // the shortest distance from src to i

        // sptSet[i] will true if vertex i is included in shortest
        // path tree or shortest distance from src to i is finalized
        Boolean sptSet[] = new Boolean[V];

        // Initialize all distances as INFINITE and stpSet[] as false
        for (int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
```

```
        }

        // Distance of source vertex from itself is always 0
        dist[src] = 0;

        // Find shortest path for all vertices
        for (int count = 0; count < V - 1; count++) {
            // Pick the minimum distance vertex from the set of vertices
            // not yet processed. u is always equal to src in first
            // iteration.
            int u = minDistance(dist, sptSet);

            // Mark the picked vertex as processed
            sptSet[u] = true;

            // Update dist value of the adjacent vertices of the
            // picked vertex.
            for (int v = 0; v < V; v++)

                // Update dist[v] only if is not in sptSet, there is an
                // edge from u to v, and total weight of path from src to
                // v through u is smaller than current value of dist[v]
                if (!sptSet[v] && graph[u][v] != 0 &&
                   dist[u] != Integer.MAX_VALUE && dist[u] + graph[u][v] < d
                    dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist, V);
    }

    // Driver method
    public static void main(String[] args)
    {
        /* Let us create the example graph discussed above */
        int graph[][] = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                      { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                      { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                      { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                      { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                      { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                      { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                      { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                      { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
        ShortestPath t = new ShortestPath();
        t.dijkstra(graph, 0);
    }
}
```

```
Vertex   Distance from Source

0                0

1                4

2                12

3                19

4                21

5                11

6                9

7                8

8                14
```

Time Complexity: The time complexity of the Dijkstra's Algorithm when implemented using a min heap is O(E * logV), where E is the number of Edges and V is the number of vertices.

Note: The Dijkstra's Algorithm **doesn't work** in the case when the Graph has negative edge weight.

− Bellman-Ford Algorithm for Shortest Path

Problem: Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

Algorithm: Following are the detailed steps.

- *Input:* Graph and a source vertex *src*.

- *Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1. This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
   Do following for each edge u-v:
   - If dist[v] > dist[u] + weight of edge uv, then update dist[v] as: *dist[v] = dist[u] + weight of edge uv*.
3. This step reports if there is a negative weight cycle in graph. Do following for each edge u-v. If dist[v] > dist[u] + weight of edge **uv**, then "Graph contains negative weight cycle".

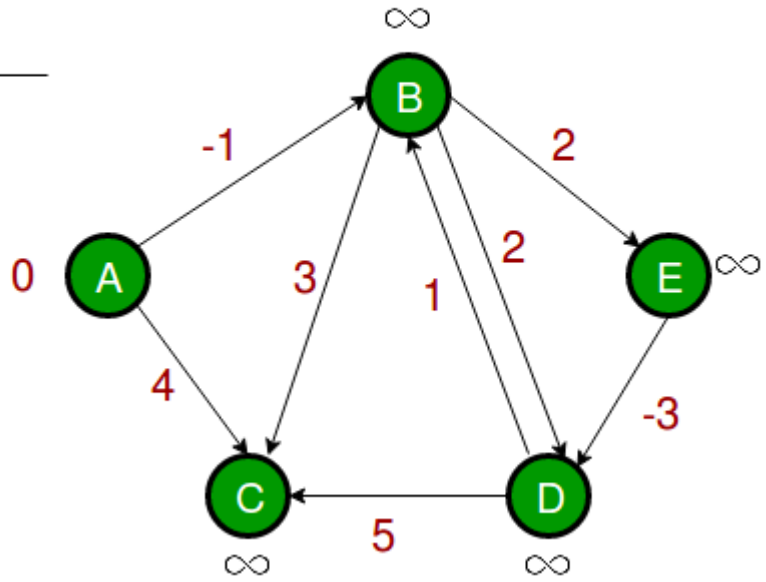
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

How does this work? Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most **i** edges are calculated. There can be maximum **|V| - 1** edge in any simple path, that is why the outer loop runs |v| - 1 time. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edge.

Example:
Let us understand the algorithm with following example graph. The images are taken from this
(http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf)source.

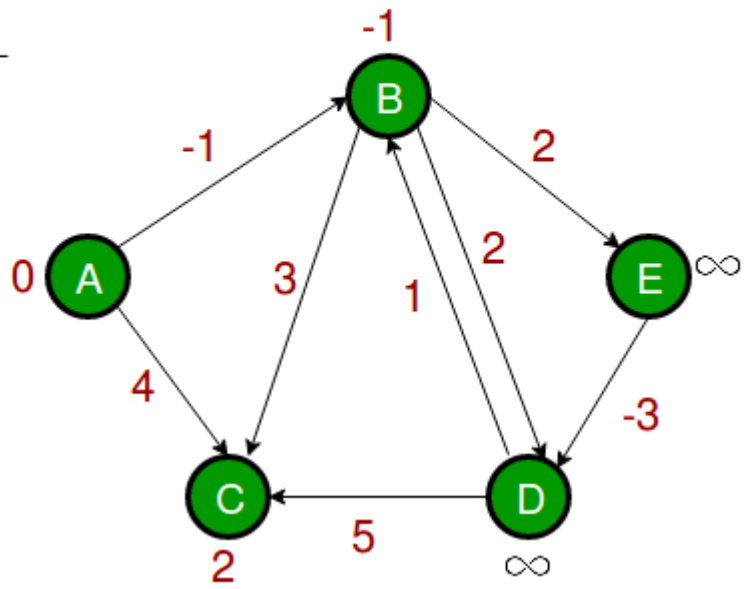
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

A	B	C	D	E
0	∞	∞	∞	∞



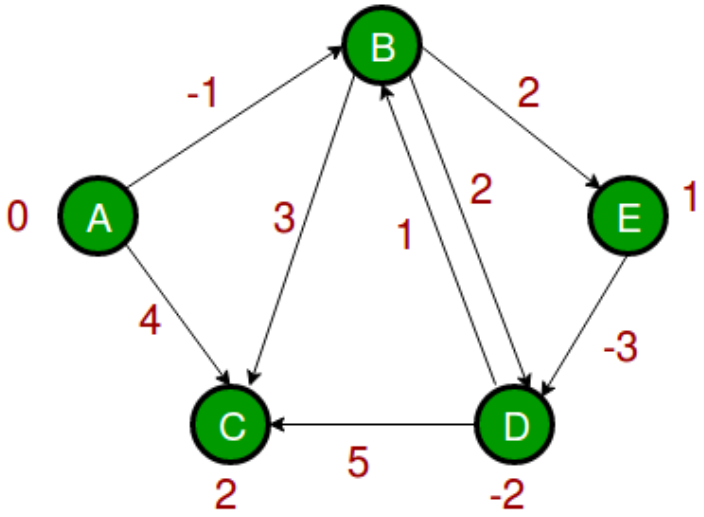
Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed a second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Implementation:

C++

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
```

Run

Java

Complete Code/Output:

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm.  The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];
```

```
    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles.  The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle.  If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }

    printArr(dist, V);

    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5;  // Number of vertices in graph
    int E = 8;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
```

```
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}
```

```
// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };

    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm.  The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];

        // Step 1: Initialize distances from src to all other
        // vertices as INFINITE
        for (int i=0; i<V; ++i)
            dist[i] = Integer.MAX_VALUE;
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i=1; i<V; ++i)
```

```
        {
            for (int j=0; j<E; ++j)
            {
                int u = graph.edge[j].src;
                int v = graph.edge[j].dest;
                int weight = graph.edge[j].weight;
                if (dist[u]!=Integer.MAX_VALUE &&
                    dist[u]+weight<dist[v])
                    dist[v]=dist[u]+weight;
            }
        }

        // Step 3: check for negative-weight cycles.  The above
        // step guarantees shortest distances if graph doesn't
        // contain negative weight cycle. If we get a shorter
        //  path, then there is a cycle.
        for (int j=0; j<E; ++j)
        {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u] != Integer.MAX_VALUE &&
                dist[u]+weight < dist[v])
              System.out.println("Graph contains negative weight cycle");
        }
        printArr(dist, V);
    }

    // A utility function used to print the solution
    void printArr(int dist[], int V)
    {
        System.out.println("Vertex   Distance from Source");
        for (int i=0; i<V; ++i)
            System.out.println(i+"\t\t"+dist[i]);
    }

    // Driver method to test above function
    public static void main(String[] args)
    {
        int V = 5;  // Number of vertices in graph
        int E = 8;  // Number of edges in graph

        Graph graph = new Graph(V, E);

        // add edge 0-1 (or A-B in above figure)
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = -1;
```

```
        // add edge 0-2 (or A-C in above figure)
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 4;

        // add edge 1-2 (or B-C in above figure)
        graph.edge[2].src = 1;
        graph.edge[2].dest = 2;
        graph.edge[2].weight = 3;

        // add edge 1-3 (or B-D in above figure)
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 2;

        // add edge 1-4 (or A-E in above figure)
        graph.edge[4].src = 1;
        graph.edge[4].dest = 4;
        graph.edge[4].weight = 2;

        // add edge 3-2 (or D-C in above figure)
        graph.edge[5].src = 3;
        graph.edge[5].dest = 2;
        graph.edge[5].weight = 5;

        // add edge 3-1 (or D-B in above figure)
        graph.edge[6].src = 3;
        graph.edge[6].dest = 1;
        graph.edge[6].weight = 1;

        // add edge 4-3 (or E-D in above figure)
        graph.edge[7].src = 4;
        graph.edge[7].dest = 3;
        graph.edge[7].weight = -3;

        graph.BellmanFord(graph, 0);
    }
}
```

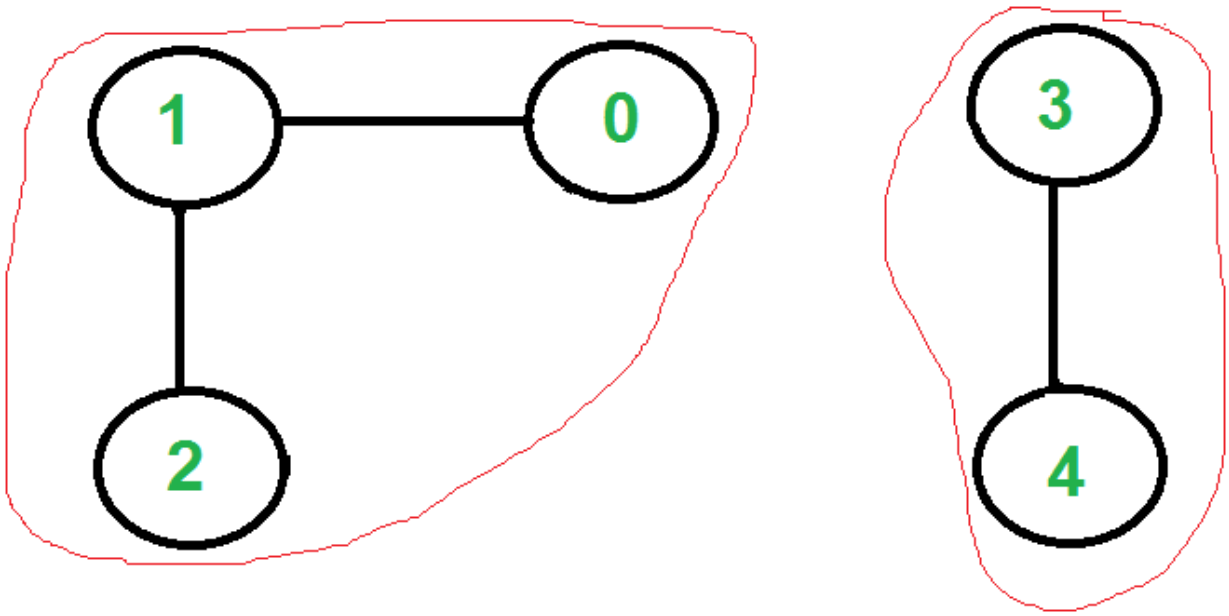
```
Vertex   Distance from Source
0                0
1                -1
2                2
3                -2
4                1
```

Important Notes:

1. Negative weights are found in various applications of graphs. For example, instead of paying the cost for a path, we may get some advantage if we follow the path.
2. Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijksra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

− Number of Strongly Connected Components in an Undirected Graph

Problem: Given an Undirected Graph. The task is to find the count of the number of *strongly connected components* in the given Graph. A **Strongly Connected Component** is defined as a subgraph of this graph in which every pair of vertices has a path in between.



There are two connected components in above undirected graph
0 1 2
3 4

Finding the connected components for an undirected graph is an easier task. The idea is to traverse all of the unvisited vertices, and for each unvisited vertex print, it's DFS or BFS traversal.

Below is the algorithm following the DFS traversal to find all connected components in an undirected graph:

```
1) Initialize all vertices as not visited.
2) Do following for every vertex 'v'.
       (a) If 'v' is not visited before, call DFSUtil(v)
       (b) Print new line character

// This Function performs DFS traversal
// of vertex v.
DFSUtil(v)
1) Mark 'v' as visited.
2) Print 'v'
3) Do following for every adjacent 'u' of 'v'.
     If 'u' is not visited, then recursively call DFSUtil(u)
```

Implementation:

C++

```
 1
 2  // C++ program to print connected components in
 3  // an undirected graph
 4  #include <iostream>
 5  #include <list>
 6  using namespace std;
 7
 8  // Graph class represents a undirected graph
 9  // using adjacency list representation
10  class Graph {
11      int V; // No. of vertices
12
13      // Pointer to an array containing adjacency lists
14      list<int>* adj;
15
16      // A function used by DFS
17      void DFSUtil(int v, bool visited[]);
18
19  public:
20      Graph(int V); // Constructor
21      void addEdge(int v, int w);
22      void connectedComponents();
23  };
24
25  // Method to print connected components in an
26  // undirected graph
27  void Graph::connectedComponents()
28  {
29      // Mark all the vertices as not visited
30      bool* visited = new bool[V];
```

Run

Java

Complete Code/Output:

```
// C++ program to print connected components in
// an undirected graph
#include <iostream>
#include <list>
using namespace std;

// Graph class represents a undirected graph
// using adjacency list representation
class Graph {
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int>* adj;

    // A function used by DFS
    void DFSUtil(int v, bool visited[]);

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w);
    void connectedComponents();
};

// Method to print connected components in an
// undirected graph
void Graph::connectedComponents()
{
    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for (int v = 0; v < V; v++)
        visited[v] = false;

    for (int v = 0; v < V; v++) {
        if (visited[v] == false) {
            // print all reachable vertices
            // from v
            DFSUtil(v, visited);

            cout << "\n";
        }
    }
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";
```

```
    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// method to add an undirected edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);
}

// Drive program to test above
int main()
{
    // Create a graph given in the above diagram
    Graph g(5); // 5 vertices numbered from 0 to 4
    g.addEdge(1, 0);
    g.addEdge(1, 2);
    g.addEdge(3, 4);

    cout << "Following are connected components \n";
    g.connectedComponents();

    return 0;
}
```

```
// Java program to print connected components in
// an undirected graph
import java.util.LinkedList;
class Graph {
    // A user define class to represent a graph.
    // A graph is an array of adjacency lists.
    // Size of array will be V (number of vertices
    // in graph)
    int V;
    LinkedList<Integer>[] adjListArray;

    // constructor
    Graph(int V)
    {
        this.V = V;
        // define the size of array as
        // number of vertices
        adjListArray = new LinkedList[V];

        // Create a new list for each vertex
        // such that adjacent nodes can be stored

        for (int i = 0; i < V; i++) {
            adjListArray[i] = new LinkedList<Integer>();
        }
    }

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

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

    // Driver program to test above
    public static void main(String[] args)
    {
        // Create a graph given in the above diagram
        Graph g = new Graph(5); // 5 vertices numbered from 0 to 4

        g.addEdge(1, 0);
        g.addEdge(1, 2);
        g.addEdge(3, 4);
        System.out.println("Following are connected components");
        g.connectedComponents();
    }
}
```

```
Following are connected components
0 1 2
3 4
```

- Prim's Minimum Spanning Tree Algorithm

What is Minimum Spanning Tree?

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A **minimum spanning tree (MST)** or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Number of edges in a minimum spanning tree: A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Prim's Algorithm

Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph theory (http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*
How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

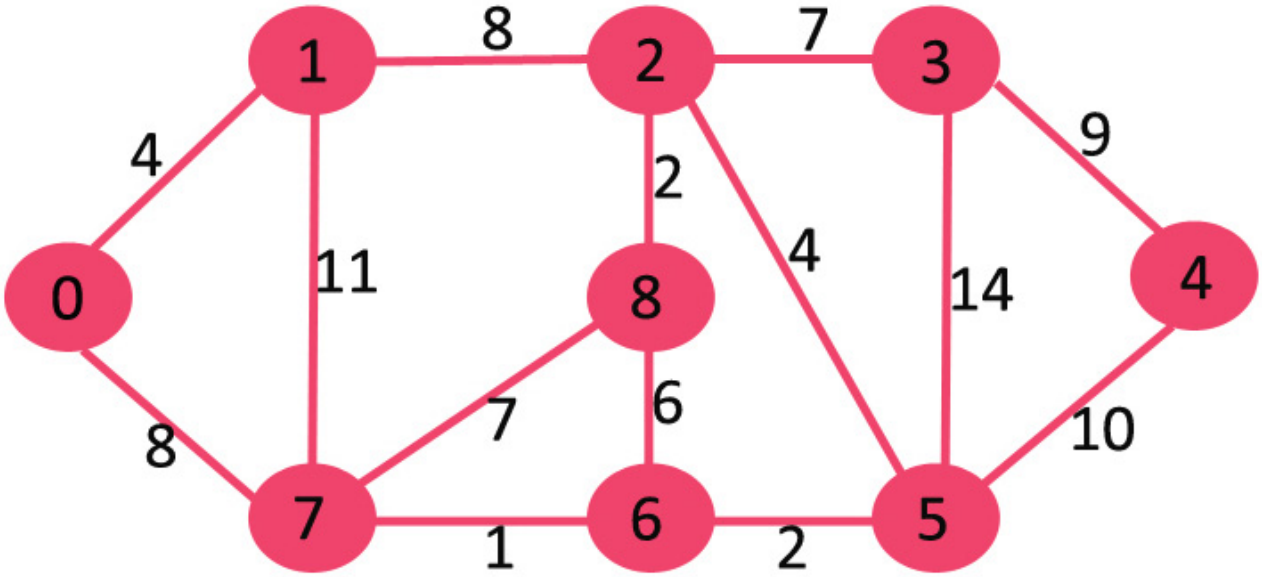
Algorithm:

1. Create a set *mstSet* that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3. While mstSet doesn't include all vertices:
    - Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
    - Include *u* to mstSet.
    - Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*.

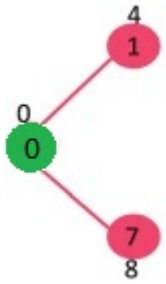
The idea of using key values is to pick the minimum weight edge from cut (http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand this with the help of following example:



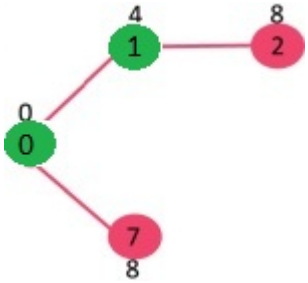
(https://www.cdn.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



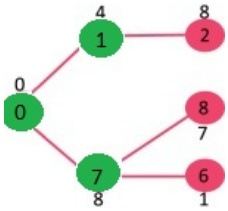
(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

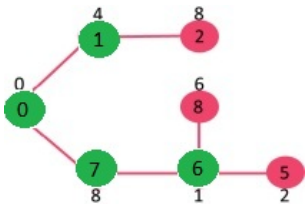


(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST2.jpg)

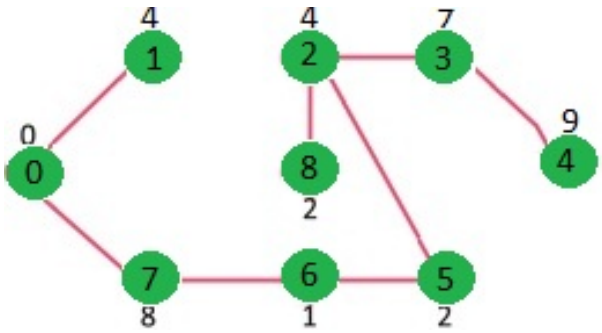
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).



(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST3.jpg)
Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST4.jpg)
We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST5.jpg)

How to implement the above algorithm?

We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C++

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
```

```
13  // not yet included in MST
14  int minKey(int key[], bool mstSet[])
15  {
16      // Initialize min value
17      int min = INT_MAX, min_index;
18
19      for (int v = 0; v < V; v++)
20          if (mstSet[v] == false && key[v] < min)
21              min = key[v], min_index = v;
22
23      return min_index;
24  }
25
26  // A utility function to print the
27  // constructed MST stored in parent[]
28  int printMST(int parent[], int graph[V][V])
29  {
30      cout<<"Edge \tWeight\n";
```

Run

Java

Complete Code/Output:

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices not yet included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
```

```
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver code
int main()
{
    /* Let us create the following graph
        2 3
    (0)--(1)--(2)
    | / \ |
    6| 8/ \5 |7
    | / \ |
    (3)-------(4)
            9     */
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
```

```
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

```
// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

class MST {
    // Number of vertices in the graph
    private static final int V = 5;

    // A utility function to find the vertex with minimum key
    // value, from the set of vertices not yet included in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min) {
                min = key[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed MST stored in
    // parent[]
    void printMST(int parent[], int graph[][])
    {
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++)
            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parer
    }

    // Function to construct and print MST for a graph represented
    // using adjacency matrix representation
    void primMST(int graph[][])
    {
        // Array to store constructed MST
        int parent[] = new int[V];

        // Key values used to pick minimum weight edge in cut
        int key[] = new int[V];

        // To represent set of vertices not yet included in MST
        Boolean mstSet[] = new Boolean[V];
```

```
    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is
    // picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick thd minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent
        // vertices of the picked vertex. Consider only those
        // vertices which are not yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] <
                parent[v] = u;
                key[v] = graph[u][v];
            }
    }

    // print the constructed MST
    printMST(parent, graph);
}

public static void main(String[] args)
{
    /* Let us create the following graph
    2 3
    (0)--(1)--(2)
    | / \ |
    6| 8/ \5 |7
    | /     \ |
    (3)-------(4)
```

```
            9         */
        MST t = new MST();
        int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
                                      { 2, 0, 3, 8, 5 },
                                      { 0, 3, 0, 0, 7 },
                                      { 6, 8, 0, 0, 9 },
                                      { 0, 5, 7, 9, 0 } };

        // Print the solution
        t.primMST(graph);
    }
}
```

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

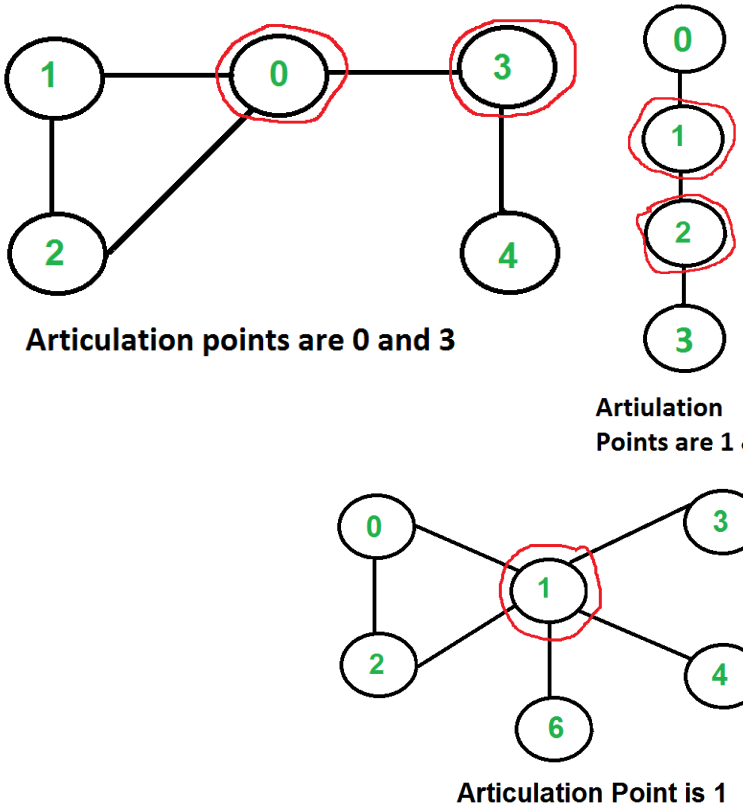
Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list (https://www.cdn.geeksforgeeks.org/archives/27134), then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap.

- Articulation Points (or Cut Vertices) in a Graph

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

For a disconnected undirected graph, an articulation point is a vertex removing which increases the number of connected components.

Following are some example graphs with articulation points encircled with red color.
(https://media.geeksforgeeks.org/wp-content/cdn-uploads/ArticulationPoints.png)
(https://media.geeksforgeeks.org/wp-content/cdn-uploads/ArticulationPoints1.png)



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/ArticulationPoints21.png)

How to find all articulation points in a given graph?

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of a simple approach for the connected graph.

1. For every vertex v, do following:
    - Remove v from graph
    - See if the graph remains connected (We can either use BFS or DFS)
    - Add v back to the graph

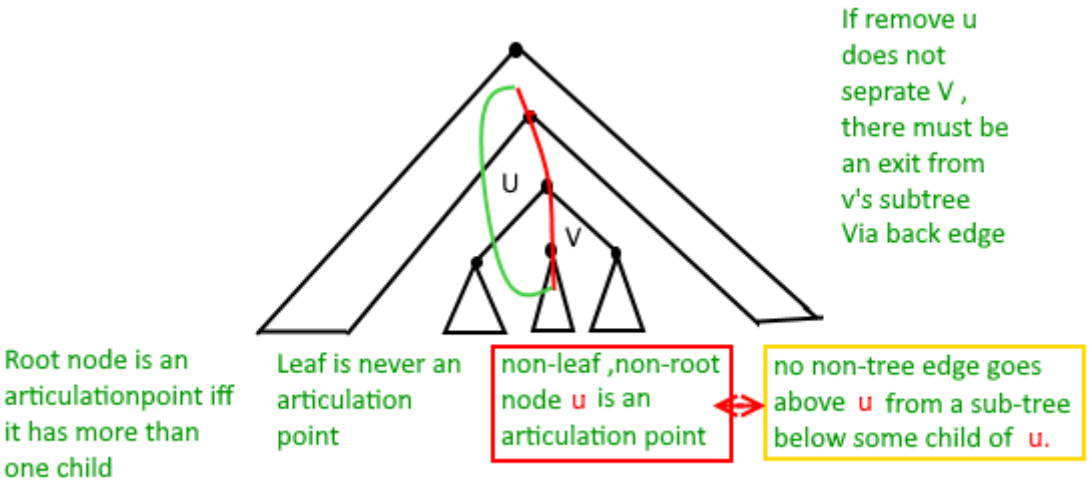
The **time complexity** of the above method is O(V*(V+E)) for a graph represented using adjacency list. Can we do better?

A O(V+E) algorithm to find all Articulation Points (APs). The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u (obviously v is adjacent of u in the graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1. u is the root of DFS tree and it has at least two children.

2. u is not the root of DFS tree and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

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



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

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

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Following are C++ and Java implementation of Tarjan's algorithm for finding articulation points:

C++

```
1
2  // A C++ program to find articulation points
3  // in an undirected graph
4  #include<iostream>
5  #include <list>
6  #define NIL -1
7  using namespace std;
8
9  // A class that represents an undirected graph
```

```
10  class Graph
11  {
12      int V;    // No. of vertices
13      list<int> *adj;    // A dynamic array of adjacency lists
14      void APUtil(int v, bool visited[], int disc[], int low[]
15                  int parent[], bool ap[]);
16  public:
17      Graph(int V);   // Constructor
18      void addEdge(int v, int w);   // function to add an edge
19      void AP();    // prints articulation points
20  };
21
22  Graph::Graph(int V)
23  {
24      this->V = V;
25      adj = new list<int>[V];
26  }
27
28  void Graph::addEdge(int v, int w)
29  {
30      adj[v].push back(w);
```

Run

Java

Complete Code/Output:

```
// A C++ program to find articulation points
// in an undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void APUtil(int v, bool visited[], int disc[], int low[],
                int parent[], bool ap[]);
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function to add an edge to graph
    void AP();    // prints articulation points
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}

// A recursive function that find articulation points using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Store articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
                                       int low[], int parent[], bool ap[])
{
    // A static variable is used for simplicity,
    // we can avoid the use of static
    // variable by passing a pointer
    static int time = 0;

    // 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 aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;  // 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 (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

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

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
               ap[u] = true;

            // (2) If u is not root and low value of one of its child is mor
            // than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
               ap[u] = true;
        }

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

// The function to do DFS traversal.
// It uses recursive function APUtil()
void Graph::AP()
{
    // Mark all the vertices as not visited
```

```
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    bool *ap = new bool[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point) arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

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

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
            cout << i << " ";
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nArticulation points in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.AP();

    cout << "\nArticulation points in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.AP();

    cout << "\nArticulation points in third graph \n";
    Graph g3(7);
```

```
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.AP();

    return 0;
}
```

```
// A Java program to find articulation points
// in an undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using
// adjacency list representation
class Graph
{
    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
    Graph(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 find articulation points using DFS
    // u --> The vertex to be visited next
    // visited[] --> keeps tract of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    // ap[] --> Store articulation points
    void APUtil(int u, boolean visited[], int disc[],
                int low[], int parent[], boolean ap[])
    {

        // 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 aadjacent 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 (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

            // 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]);

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
                ap[u] = true;

            // (2) If u is not root and low value of one of its child
            // is more than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
                ap[u] = true;
        }

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

// The function to do DFS traversal.
// It uses recursive function APUtil()
void AP()
{
    // 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];
        boolean ap[] = new boolean[V]; // To store articulation points

        // Initialize parent and visited, and ap(articulation point)
        // arrays
        for (int i = 0; i < V; i++)
        {
            parent[i] = NIL;
            visited[i] = false;
            ap[i] = false;
        }

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

        // Now ap[] contains articulation points, print them
        for (int i = 0; i < V; i++)
            if (ap[i] == true)
                System.out.print(i+" ");
    }

    // Driver method
    public static void main(String args[])
    {
        // Create graphs given in above diagrams
        System.out.println("Articulation points in first graph ");
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        g1.AP();
        System.out.println();

        System.out.println("Articulation points in Second graph");
        Graph g2 = new Graph(4);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        g2.addEdge(2, 3);
        g2.AP();
        System.out.println();

        System.out.println("Articulation points in Third graph ");
        Graph g3 = new Graph(7);
```

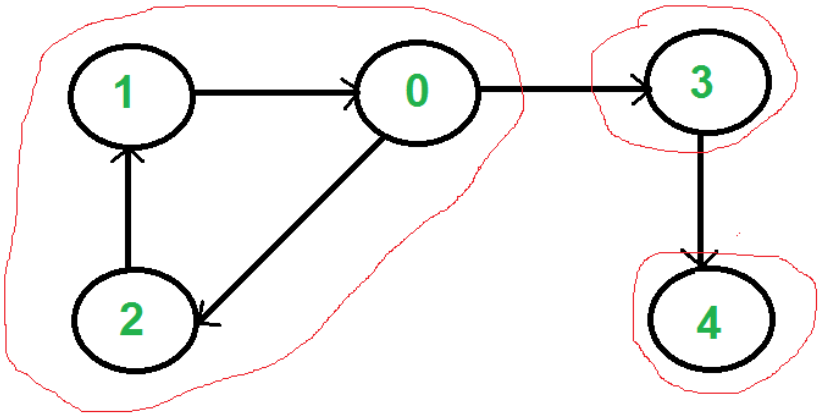
```
        g3.addEdge(0, 1);
        g3.addEdge(1, 2);
        g3.addEdge(2, 0);
        g3.addEdge(1, 3);
        g3.addEdge(1, 4);
        g3.addEdge(1, 6);
        g3.addEdge(3, 5);
        g3.addEdge(4, 5);
        g3.AP();
    }
}
```

```
Articulation points in first graph
0 3
Articulation points in second graph
1 2
Articulation points in third graph
1
```

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is the same as DFS which is O(V+E) for adjacency list representation of the graph.

- Kosaraju's Algorithm | Strongly Connected Components

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



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/SCC.png)
We can find all strongly connected components in O(V+E) time using Kosaraju's

algorithm (http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm). Following is detailed Kosaraju's algorithm.

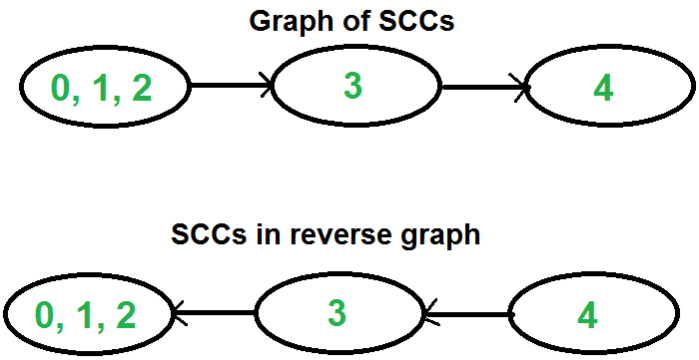
1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in the stack as 1, 2, 4, 3, 0.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v) (https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise, DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start at 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sunk in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way of getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC (See this (http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/strongComponent.htm)for proof). For example, in DFS of above example graph, the finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And the finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example, finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of the complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.

In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in the stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in the stack, we process vertices from the sink to the source (in the reversed

graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/SCCGraph2.png)

Below is the implementation of Kosaraju's algorithm:

C++

```
// C++ Implementation of Kosaraju's algorithm to print all S
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // An array of adjacency lists

    // Fills Stack with vertices (in increasing order of fir
    // times). The top element of stack has the maximum fini
    // time
    void fillOrder(int v, bool visited[], stack<int> &Stack)

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);
    void addEdge(int v, int w);

    // The main function that finds and prints strongly conr
    // components
    void printSCCs();

    // Function that returns reverse (or transpose) of this
    Graph getTranspose();
};
```

Run

Java

Complete Code/Output:

```
// C++ Implementation of Kosaraju's algorithm to print all SCCs
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // An array of adjacency lists

    // Fills Stack with vertices (in increasing order of finishing
    // times). The top element of stack has the maximum finishing
    // time
    void fillOrder(int v, bool visited[], stack<int> &Stack);

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);
    void addEdge(int v, int w);

    // The main function that finds and prints strongly connected
    // components
    void printSCCs();

    // Function that returns reverse (or transpose) of this graph
    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
```

```
}

Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);

    // All vertices reachable from v are processed by now, push v
    Stack.push(v);
}

// The main function that finds and prints all
// strongly connected components
void Graph::printSCCs()
{
    stack<int> Stack;

    // Mark all the vertices as not visited (For first DFS)
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;
```

```
    // Fill vertices in stack according to their finishing times
    for(int i = 0; i < V; i++)
        if(visited[i] == false)
            fillOrder(i, visited, Stack);

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Now process all vertices in order defined by Stack
    while (Stack.empty() == false)
    {
        // Pop a vertex from stack
        int v = Stack.top();
        Stack.pop();

        // Print Strongly connected component of the popped vertex
        if (visited[v] == false)
        {
            gr.DFSUtil(v, visited);
            cout << endl;
        }
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "Following are strongly connected components in "
            "given graph \n";
    g.printSCCs();

    return 0;
}
```

```
// Java implementation of Kosaraju's algorithm to print all SCCs

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

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(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); }

    // A recursive function to print DFS starting from v
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v + " ");

        int n;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i =adj[v].iterator();
        while (i.hasNext())
        {
            n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // Function that returns reverse (or transpose) of this graph
    Graph getTranspose()
    {
        Graph g = new Graph(V);
```

```
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i =adj[v].listIterator();
        while(i.hasNext())
            g.adj[i.next()].add(v);
    }
    return g;
}

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

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i = adj[v].iterator();
    while (i.hasNext())
    {
        int n = i.next();
        if(!visited[n])
            fillOrder(n, visited, stack);
    }

    // All vertices reachable from v are processed by now,
    // push v to Stack
    stack.push(new Integer(v));
}

// The main function that finds and prints all strongly
// connected components
void printSCCs()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited (For first DFS)
    boolean visited[] = new boolean[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Fill vertices in stack according to their finishing
    // times
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            fillOrder(i, visited, stack);

    // Create a reversed graph
    Graph gr = getTranspose();
```

```
        // Mark all the vertices as not visited (For second DFS)
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Now process all vertices in order defined by Stack
        while (stack.empty() == false)
        {
            // Pop a vertex from stack
            int v = (int)stack.pop();

            // Print Strongly connected component of the popped vertex
            if (visited[v] == false)
            {
                gr.DFSUtil(v, visited);
                System.out.println();
            }
        }
    }

    // Driver method
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g = new Graph(5);
        g.addEdge(1, 0);
        g.addEdge(0, 2);
        g.addEdge(2, 1);
        g.addEdge(0, 3);
        g.addEdge(3, 4);

        System.out.println("Following are strongly connected components "+
                           "in given graph ");
        g.printSCCs();
    }
}
// This code is contributed by Aakash Hasija
```

```
Following are strongly connected components in given graph

0 1 2

3

4
```

Time Complexity: The above algorithm calls DFS, fins reverse of the graph and again

calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simply traverse all adjacency lists.

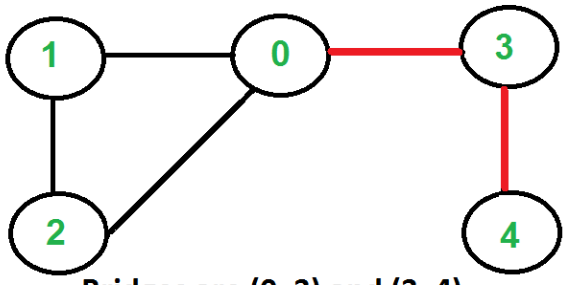
Applications: SCC algorithms can be used as the first step in many graph algorithms that work only on a strongly connected graph. In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

− Bridges in a Graph

An edge in an undirected connected graph is a bridge *if and only if* removing it disconnects the graph. For a disconnected undirected graph, the definition is similar, a bridge is an edge removing which increases the number of disconnected components.

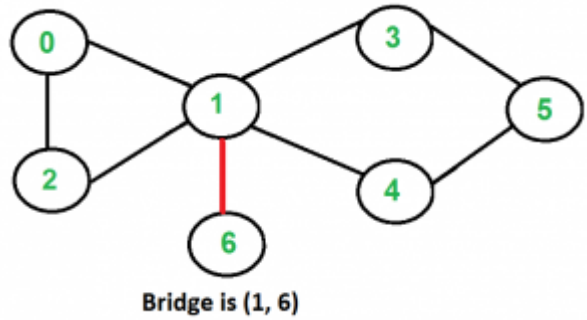
Like ***Articulation Points***, bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red colour:



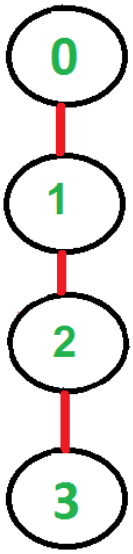
Bridges are (0, 3) and (3, 4)

(https://media.geeksforgeeks.org/wp-content/cdn-uploads/Bridge1.png)



Bridge is (1, 6)

(https://media.geeksforgeeks.org/wp-content/cdn-uploads/Bridge2.png)



**Bridges are (0,1),
(1,2) and (2,3)**

(https://media.geeksforgeeks.org/wp-content/cdn-uploads/Bridge3.png)

How to find all bridges in a given graph?

A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of a simple approach for a connected graph.

1. For every edge (u, v), do following
    - Remove (u, v) from graph.
    - See if the graph remains connected (We can either use BFS or DFS)
    - Add (u, v) back to the graph.

The **time complexity** of the above method is O(E*(V+E)) for a graph represented using adjacency list. Can we do better?

A O(V+E) algorithm to find all Bridges is similar to that of O(V+E) algorithm for Articulation Points. We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the previous post, the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, "low[v] > disc[u]".*

Following are C++ and Java implementations of the above approach:

C++

```
1
2  // A C++ program to find bridges in a given undirected graph
3
4  #include<iostream>
5  #include <list>
6  #define NIL -1
```

```
 7  using namespace std;
 8
 9  // A class that represents an undirected graph
10  class Graph
11  {
12      int V;    // No. of vertices
13      list<int> *adj;    // A dynamic array of adjacency lists
14      void bridgeUtil(int v, bool visited[], int disc[], int l
15                      int parent[]);
16  public:
17      Graph(int V);   // Constructor
18      void addEdge(int v, int w);   // to add an edge to graph
19      void bridge();    // prints all bridges
20  };
21
22  Graph::Graph(int V)
23  {
24      this->V = V;
25      adj = new list<int>[V];
26  }
27
28  void Graph::addEdge(int v, int w)
29  {
30      adj[v].push back(w);
```

Run

Java

Complete Code/Output:

```
// A C++ program to find bridges in a given undirected graph

#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void bridgeUtil(int v, bool visited[], int disc[], int low[],
                    int parent[]);
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // to add an edge to graph
    void bridge();    // prints all bridges
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                                  int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 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 aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;  // v is current adjacent of u

        // 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]  = 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])
              cout << u <<" " << v << endl;
        }

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

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited 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);
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nBridges in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();

    cout << "\nBridges in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();

    cout << "\nBridges in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.bridge();

    return 0;
}
```

```
// A Java program to find bridges in a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using
// adjacency list representation
class Graph
{
    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
    Graph(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 tract 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 aadjacent 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[])
    {
        // Create graphs given in above diagrams
        System.out.println("Bridges in first graph ");
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        g1.bridge();
        System.out.println();

        System.out.println("Bridges in Second graph");
        Graph g2 = new Graph(4);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        g2.addEdge(2, 3);
        g2.bridge();
        System.out.println();

        System.out.println("Bridges in Third graph ");
        Graph g3 = new Graph(7);
        g3.addEdge(0, 1);
        g3.addEdge(1, 2);
        g3.addEdge(2, 0);
        g3.addEdge(1, 3);
        g3.addEdge(1, 4);
        g3.addEdge(1, 6);
        g3.addEdge(3, 5);
        g3.addEdge(4, 5);
        g3.bridge();
    }
}
```

```
Bridges in first graph

3 4

0 3



Bridges in second graph

2 3

1 2

0 1



Bridges in third graph

1 6
```

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is O(V+E) for adjacency list representation of graph.

- Sample Problems on Graphs

Problem #1 : Find the number of islands

Description - Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

Example:

```
Input : mat[][] = {{1, 1, 0, 0, 0},
                   {0, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1},
                   {0, 0, 0, 0, 0},
                   {1, 0, 1, 0, 1}
Output : 5
```

Solution - The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected

components. BFS can also be used.

What is an island ? A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1,  1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
```

Pseudo Code

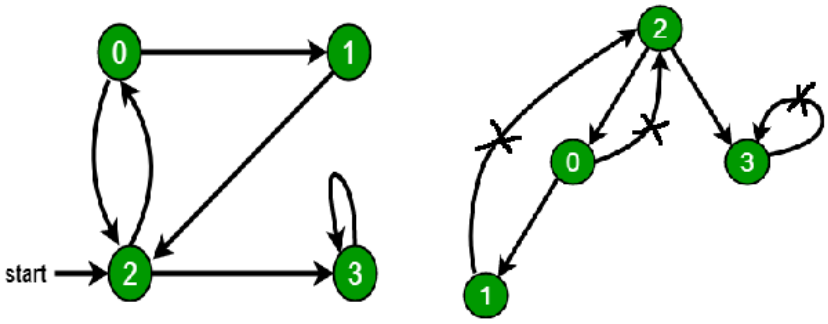
```
// A utility function to do DFS for a
// 2D boolean matrix. It only considers
// the 8 neighbours as adjacent vertices
void DFS(int M[][COL], int row, int col,
         bool visited[][COL])
{
    // These arrays are used to get
    // row and column numbers of 8
    // neighbours of a given cell
    rowNbr[] = { -1, -1, -1, 0, 0, 1, 1, 1 }
    colNbr[] = { -1, 0, 1, -1, 1, -1, 0, 1 }

    // Mark this cell as visited
    visited[row][col] = true

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited))
            DFS(M, row + rowNbr[k], col + colNbr[k], visited)
}

// The main function that returns
// count of islands in a given boolean
// 2D matrix
int countIslands(int M[][COL])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW][COL]
    memset(visited, 0, sizeof(visited))

    // Initialize count as 0 and
    // travese through the all cells of
    // given matrix
    int count = 0
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)

            // If a cell with value 1 is not
            if (M[i][j] && !visited[i][j]) {
                // visited yet, then new island found
                // Visit all cells in this island.
                DFS(M, i, j, visited)

                // and increment island count
                ++count
            }
```

```
    return count
}
```

Problem #2 : Detect Cycle in a Directed Graph

Description - Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false.

Solution - Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge (http://en.wikipedia.org/wiki/Depth-first_search#Output_of_a_depth-first_search) present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



(https://media.geeksforgeeks.org/wp-content/uploads/cycle.png)
For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for a cycle in individual trees by checking back edges.

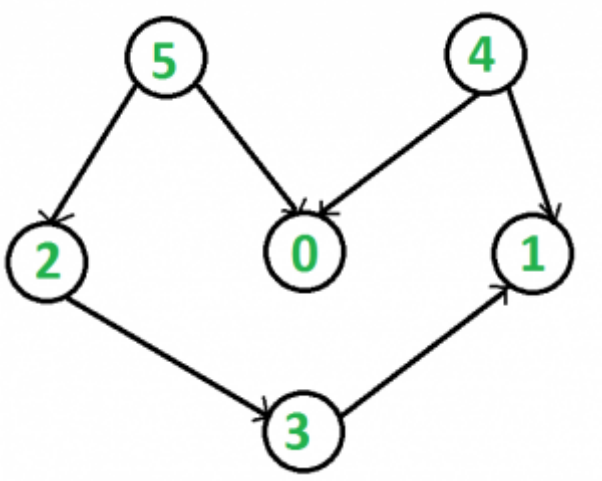
To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

Pseudo Code

```
// Utility function to check back edge in a directed graph
bool isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true
        recStack[v] = true

        // Recur for all the vertices adjacent to this vertex
        list < int > :: iterator i
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true
            else if (recStack[*i])
                return true
        }

    }
    recStack[v] = false;  // remove the vertex from recursion stack
    return false;
}

// Returns true if the graph contains a cycle, else false.
bool isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V]
    bool *recStack = new bool[V]
    for(int i = 0; i < V; i++)
    {
        visited[i] = false
        recStack[i] = false
    }

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true

    return false
}
```

Problem #3 : Topological Sorting

Description - Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.
For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Solution - We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Pseudo Code

```
// A recursive function used by topological sort
void topologicalSortUtil(int v, bool visited[],
                                 stack &Stack)
{
    // Mark the current node as visited.
    visited[v] = true

    // Recur for all the vertices adjacent to this vertex
    list < int > :: iterator i
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v)
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void topologicalSort()
{
    stack < int > Stack

    // Mark all the vertices as not visited
    bool *visited = new bool[V]
    for (int i = 0; i < V; i++)
        visited[i] = false

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
      if (visited[i] == false)
        topologicalSortUtil(i, visited, Stack)

    // Print contents of stack
    // Topological Order
    while (Stack.empty() == false)
    {
        print(Stack.top())
        Stack.pop()
    }
}
```

Problem #4 : Minimum time required to rot all oranges

Description - Given a matrix of dimension m*n where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

```
0: Empty cell
1: Cells have fresh oranges
2: Cells have rotten oranges
```

So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1, j], [i+1, j], [i, j-1], [i, j+1] (up, down, left and right). If it is impossible to rot every orange then simply return -1.

Examples

```
Input:  arr[][C] = { {2, 1, 0, 2, 1},
                     {1, 0, 1, 2, 1},
                     {1, 0, 0, 2, 1}};
Output:
All oranges can become rotten in 2 time frames.


Input:  arr[][C] = { {2, 1, 0, 2, 1},
                     {0, 0, 1, 2, 1},
                     {1, 0, 0, 2, 1}};
Output:
All oranges cannot be rotten.
```

Solution - The idea is to user Breadth First Search. Below is the algorithm.

1) Create an empty Q.
2) Find all rotten oranges and enqueue them to Q. Also enqueue a delimiter to indicate the beginning of next time frame.
3) While Q is not empty do following
....3.a) Do following while delimiter in Q is not reached
..... (i) Dequeue an orange from queue, rot all adjacent oranges. While rotting the adjacent, make sure that the time frame is incremented only once. And the time frame is not incremented if there are no adjacent oranges.
....3.b) Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in the previous time frame lie between the two delimiters.

Pseudo Code

```
// This function finds if it is possible to rot all oranges or not.
// If possible, then it returns minimum time required to rot all,
// otherwise returns -1
int rotOranges(int arr[][C])
{
    // Create a queue of cells
    queue < cell > Q
    cell temp
    ans = 0
    // Store all the cells having rotten orange in first time frame
    for (int i=0; i < R; i++)
    {
       for (int j=0; j < C; j++)
       {
            if (arr[i][j] == 2)
            {
                temp.x = i
                temp.y = j
                Q.push(temp)
            }
        }
    }
    // Separate these rotten oranges from the oranges which will rotten
    // due the oranges in first time frame using delimiter which is (-1, -1)
    temp.x = -1
    temp.y = -1
    Q.push(temp)
    // Process the grid while there are rotten oranges in the Queue
    while (!Q.empty())
    {
        // This flag is used to determine whether even a single fresh
        // orange gets rotten due to rotten oranges in current time
        // frame so we can increase the count of the required time.
        bool flag = false
        // Process all the rotten oranges in current time frame.
        while (!isdelim(Q.front()))
        {
            temp = Q.front()
            // Check right adjacent cell that if it can be rotten
            if (isvalid(temp.x+1, temp.y) && arr[temp.x+1][temp.y] == 1)
            {
                // if this is the first orange to get rotten, increase
                // count and set the flag.
                if (!flag) ans++, flag = true

                // Make the orange rotten
                arr[temp.x+1][temp.y] = 2
```

```
                // push the adjacent orange to Queue
                temp.x++
                Q.push(temp)

                temp.x-- // Move back to current cell
            }
            // Check left adjacent cell that if it can be rotten
            if (isvalid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1) {
                if (!flag) ans++, flag = true
                arr[temp.x-1][temp.y] = 2
                temp.x--
                Q.push(temp) // push this cell to Queue
                temp.x++
            }
            // Check top adjacent cell that if it can be rotten
            if (isvalid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {
                if (!flag) ans++, flag = true
                arr[temp.x][temp.y+1] = 2
                temp.y++
                Q.push(temp) // Push this cell to Queue
                temp.y--
            }
            // Check bottom adjacent cell if it can be rotten
            if (isvalid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1) {
                if (!flag) ans++, flag = true
                arr[temp.x][temp.y-1] = 2
                temp.y--
                Q.push(temp) // push this cell to Queue
            }
            Q.pop()
        }
        // Pop the delimiter
        Q.pop()
        // If oranges were rotten in current frame than separate the
        // rotten oranges using delimiter for the next frame for processing.
        if (!Q.empty()) {
            temp.x = -1
            temp.y = -1
            Q.push(temp)
        }
        // If Queue was empty than no rotten oranges left to process so exit
    }
    // Return -1 if all arranges could not rot, otherwise -1.
    return (checkall(arr))? -1: ans
}
```