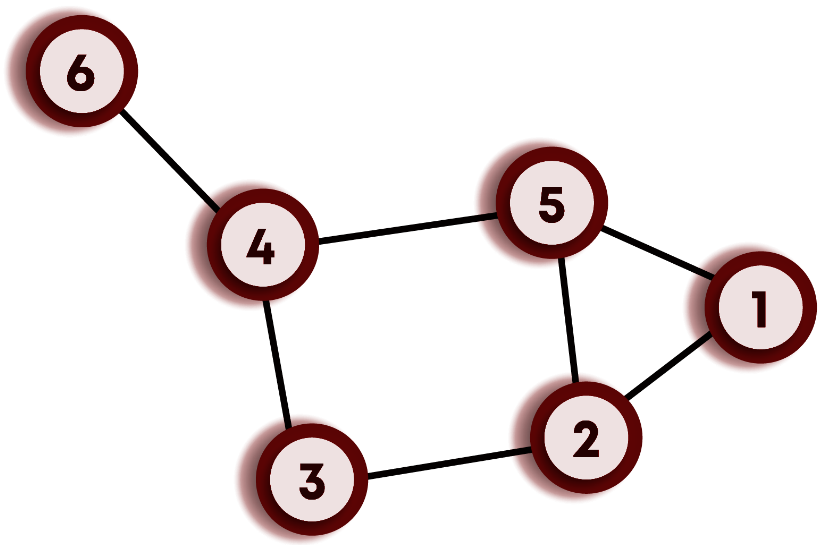
Graphs- 1

Introduction

A **graph** is a pair *G* = (*V*, *E*), where *V* is a set whose elements are called ***vertices***, and *E* is a set of two sets of vertices, whose elements are called *edges*.

The vertices *x* and *y* of an edge **{*x*, *y*}** are called the ***endpoints*** of the edge. The edge is said to ***join*** *x* and *y* and to be ***incident*** on ***x*** and ***y***. A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use trees to store it. In such cases, we will be using graphs. Refer to the figure for better understanding.



Relationship between trees and graphs:

- A tree is a special type of graph in which we can reach any node to any other node using some path, unlike the graphs where this condition may or may not hold.
- A tree does not have any cycles in it.

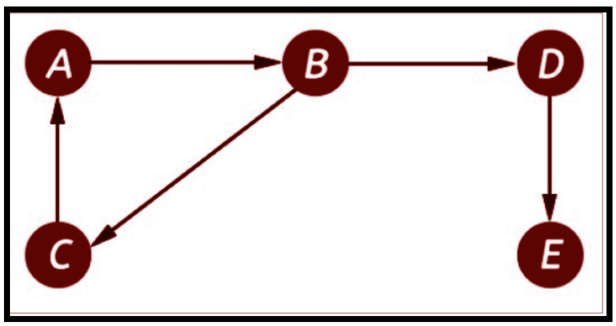
Graphs Terminology

- Nodes are named **vertices,** and the connections between them are called **edges**.
- Two vertices are said to be **adjacent** if there exists a direct edge connecting them.
- The **degree** of a node is defined as the number of edges that are incident to it.
- A **path** is a collection of edges through which we can reach from one node to the other node in a graph.
- A graph is said to be **connected** if there is a path between every pair of vertices.
- If the graph is not connected, then all the connected subsets of the graphs are called **connected components**. Each component is connected within the self, but two different components of a graph are never connected.
- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be **(N-1)**, where **N** is the number of nodes.

- In a complete graph (where each node is connected to every other node by a direct edge), there are ${}^{N}C_2$ number of edges means **(N * (N-1)) / 2** edges, where n is the number of nodes.
- This is the maximum number of edges that a graph can have.
- Hence, if an algorithm works on the terms of edges, let's say **O(E)**, where **E** is the number of edges, then in the worst case, the algorithm will take **$O(N^2)$** time, where **N** is the number of nodes.

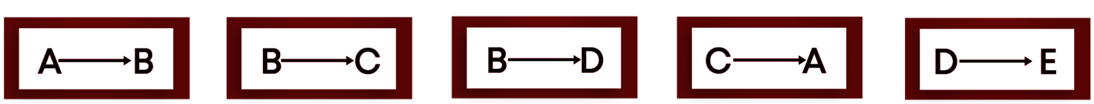
Graphs Implementation

Suppose the graph is as follows:



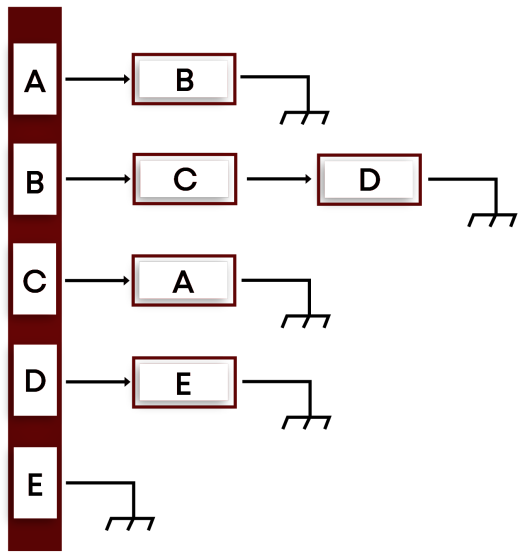
There are the following ways to implement a graph:

1. **Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred to check for a particular edge connecting two nodes; we have to traverse the complete array leading to $O(n^2)$ time complexity in the worst case. Pictorial representation for the above graph using the edge list is given below:



2. **Adjacency list:** We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. Now to check for a particular edge, we can take any one of the nodes and then check

in its list if the target node is present or not. This will take O(n) work to figure out a particular edge.



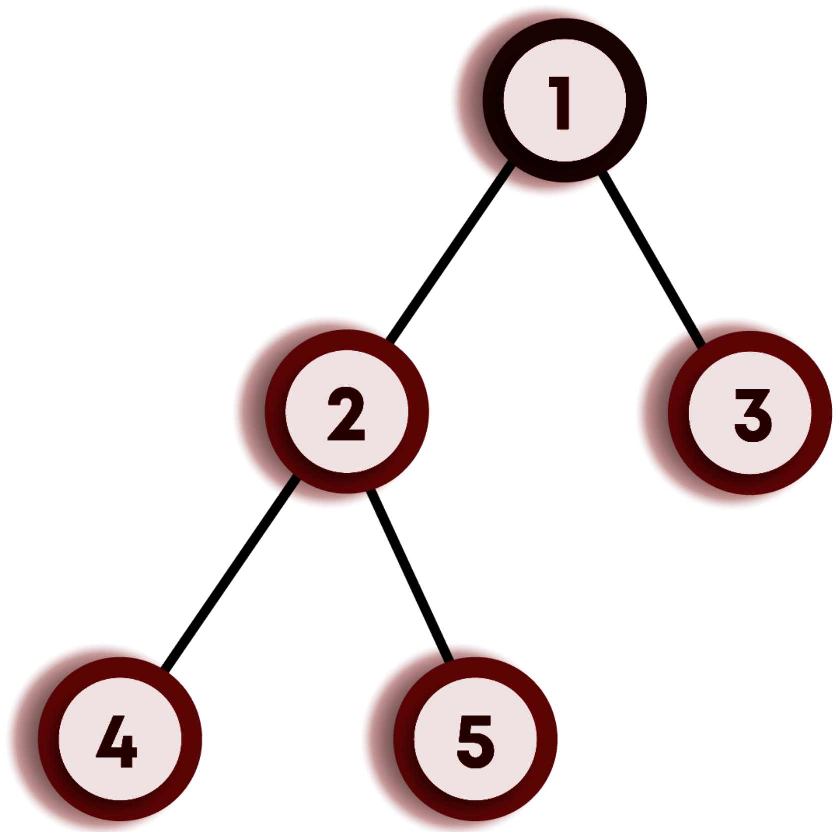
3. **Adjacency matrix:** Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j. It is the most reliable method to implement a graph in terms of ease of implementation. We will be using the same throughout the session. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each node stores only those nodes that are directly connected to them. For the above graph, the adjacency matrix looks as follows:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

DFS (Depth First Search)- Take Input and Print Graph

- Suppose we wish to create a graph containing N edges.
- We will store these edges in an Adjacency Matrix.
- We will run the loop from **0** to **number_of_edges**, and at each iteration, we will take input for the two connected nodes and correspondingly update the adjacency matrix. Let's look at the Python code for a better understanding.

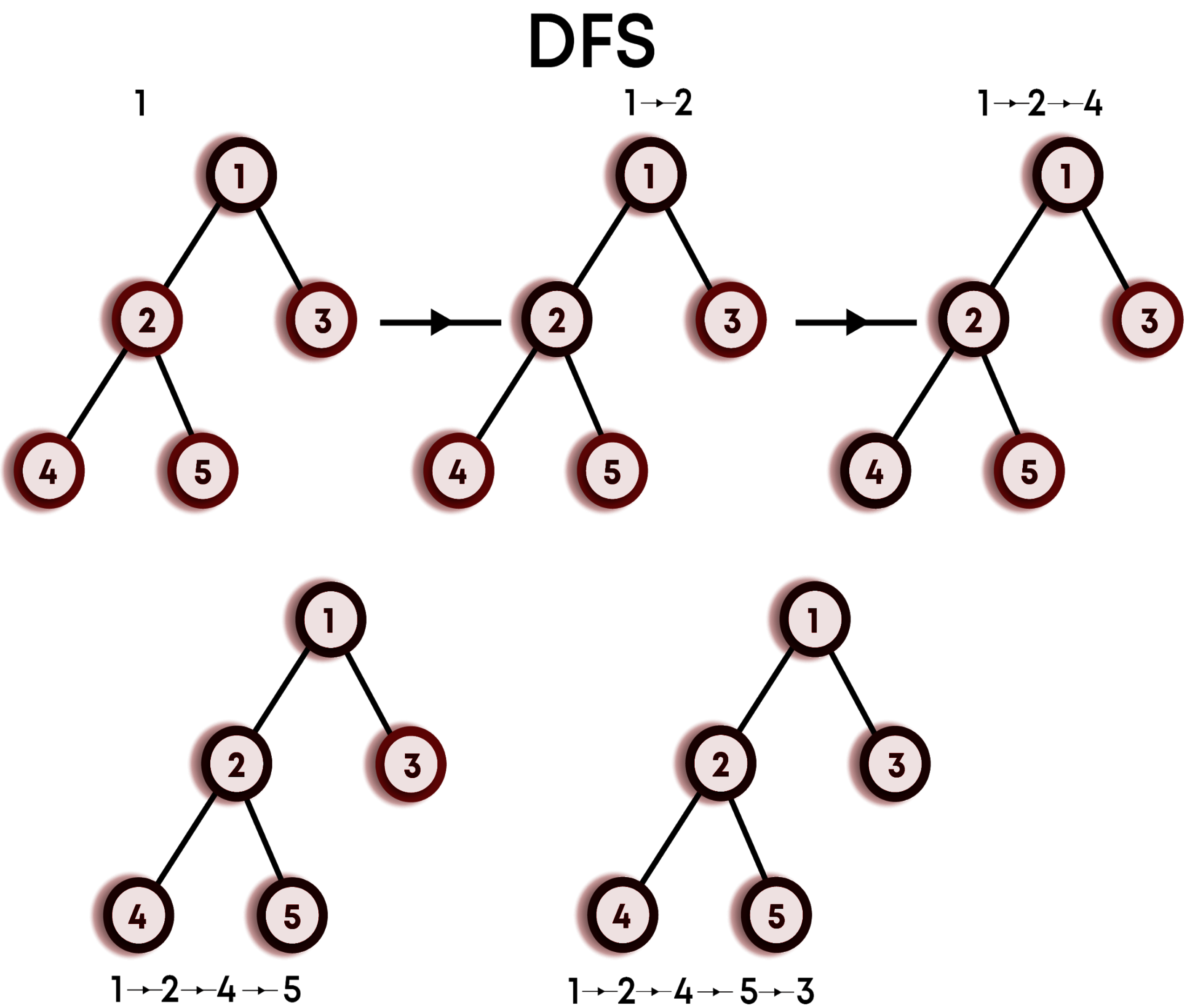
Let's take an example graph:



On dry running the above code, the output will be **1 2 4 5 3.**

Here, we are starting from a node, going in one direction as far as we can, and then we return and do the same on the previous nodes. This method of graph traversal is known as the **depth-first search (DFS)**. As the name suggests, this algorithm goes into the depth-first and then recursively does the same in other directions.

Follow the figure below, for step-by-step traversal using DFS.



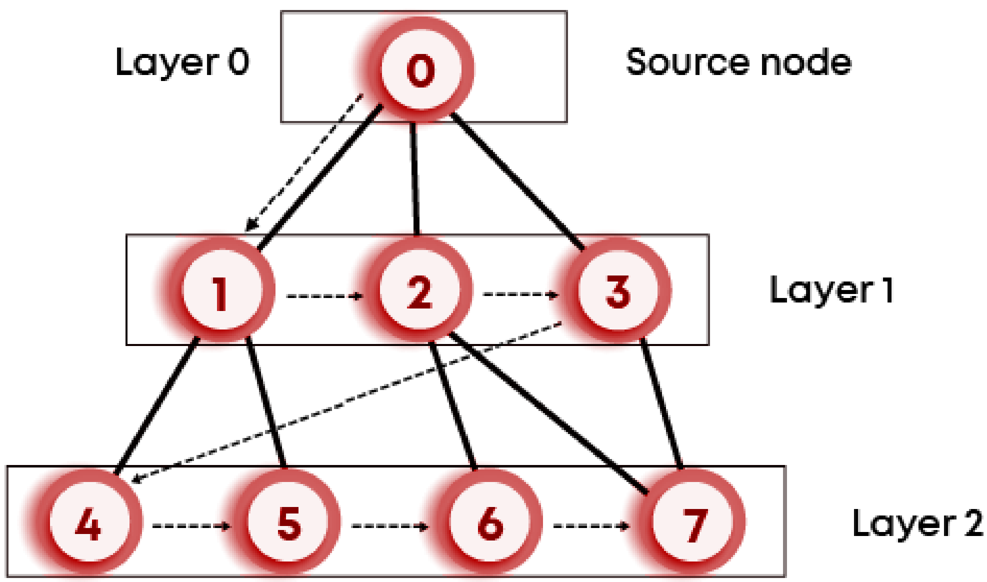
```python
class Graph:

    def __init__ (self,nVertices):
        self.nVertices = nVertices
        self.adjMatrix = [[0 for i in range(nVertices)] for j in range(nVertices)]

    def addEdge(self,v1,v2):
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    def __dfsHelper(self,sv,visited):
        print(sv)
        visited[sv]= True
        for i in range(self.nVertices):
            if(self.adjMatrix[sv][i]>0 and visited[i] is False:
                self.__dfsHelper(i, visited)
    def dfs(self):
        visited = [False for i in range(self.nVertices)]
        self.__dfsHelper(0,visited)

    def removeEdge(self,v1,v2):
        if not self.containsEdge(v1,v2):
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v2] = 0

    def containsEdge(self,v1,v2):
        return True if self.adjMatrix[v1][v2] > 0 else False
```

BFS Traversal

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

As the name suggests:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.



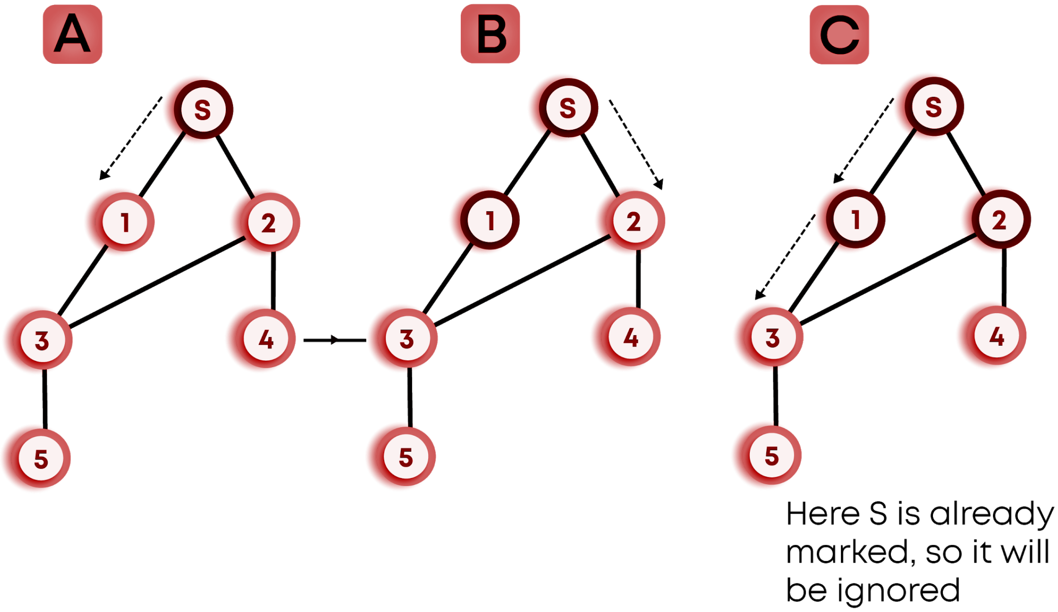
This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

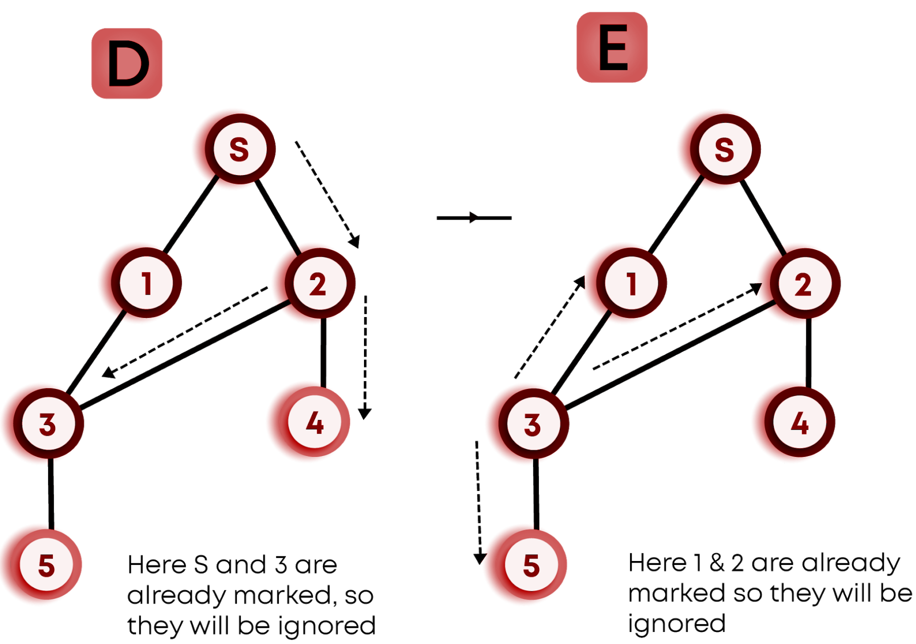
Let's look at the code below:

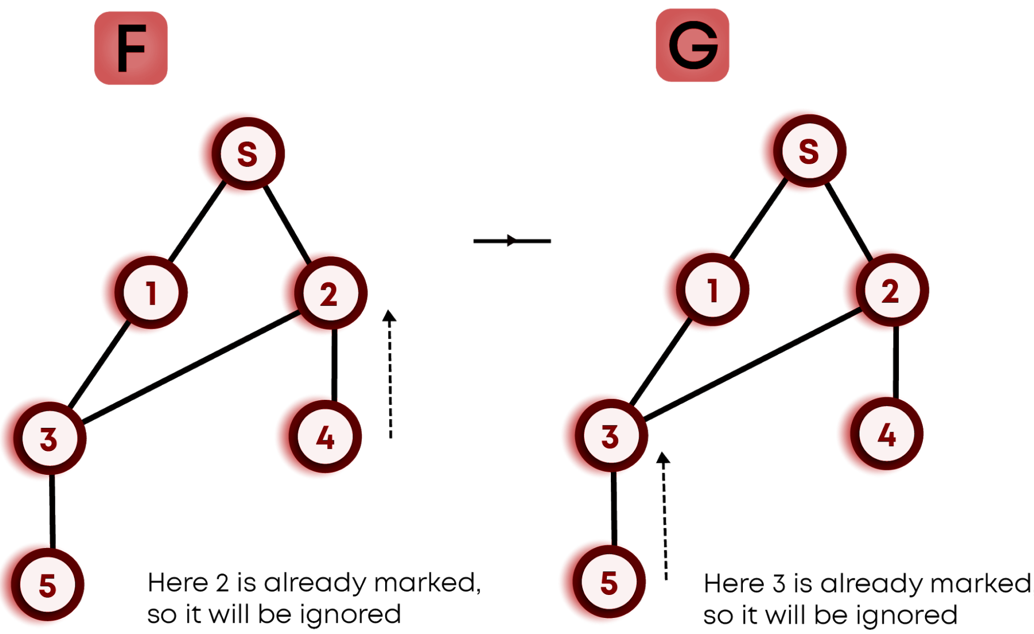
```
class Graph:

    def __init__ (self,nVertices):
        self.nVertices = nVertices
```

```
        self.adjMatrix = [[0 for i in range(nVertices)] for j in range(nVertices)]

    def addEdge(self,v1,v2):
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    def __bfs(self, sv, visited):
        q = queue. Queue()
        q.put(sv)

        visited[sv] = True

        while q.empty() is False :
            u = q.get()
            print(u, end=" ")
            for i in range(self.nVertices) :
                if self.adjMatrix[u][i] > 0 and visited[i] is False :
                    q.put(i)
                    visited[i] = True

    def bfs(self):
        visited = [False for i in range(self.nVertices)]
        for i in range(self.nVertices):
            if visited[i] is False:
                self.__bfs(i, visited)



    def removeEdge(self,v1,v2):
        if not self.containsEdge(v1,v2):
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v2] = 0

    def containsEdge(self,v1,v2):
        return True if self.adjMatrix[v1][v2] > 0 else False
```

Consider the dry run over the example graph below for a better understanding of the same:


A
B
C
S
1
2
3
4
5
Here S is already marked, so it will be ignored


D
E
S
1
2
3
4
5
Here S and 3 are already marked, so they will be ignored
Here 1 & 2 are already marked so they will be ignored



BFS & DFS for Disconnected Graph

Till now, we have assumed that the graph is connected. For the disconnected graph, there will be a minor change in the above codes. Just before calling out the print functions, we will run a loop over each node and check if that node is visited or not. If not visited, then we will call a print function over that node, considering it as the starting vertex. In this way, we will be able to cover up all the nodes of the graph.

Consider the same for the BFS function. Just replace this function in the above code to make it work for the disconnected graph too.

Has Path

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), check if there exists any path between them or not. Print true or false. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

Approach: This can be simply solved by considering the vertex v1 as the starting vertex and then run either BFS or DFS as per your choice, and while traversing if we reach the vertex v2, then we will simply return true, otherwise return false.

This problem has been left for you to try yourself. For code, refer to the solution tab of the same.

Get Path - DFS

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.
Find the path using DFS and print the first path that you encountered irrespective of the length of the path. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.
Print the path in reverse order. That is, print v2 first, then intermediate vertices, and v1 at last.
Example: Suppose the given input is:

```
4 4
0 1
```

```
0 3
1 2
2 3
1 3
```

The output should be:

```
3 0 1
```

Explanation: Here, v1 = 1 and v2 = 3. The connected vertex pairs are (0, 1), (0, 3), (1, 2) and (2, 3). So, according to the question, we have to print the path from vertex v1 to v2 in reverse order using DFS only; hence the path comes out to be {3, 0, 1}.

Approach: We have to solve this problem by using DFS. Suppose, if the start and end vertex are the same, then we simply need to put the start in the solution array and return the solution array. If this is not the case, then from the start vertex, we will call DFS on the direct connections of the same. If none of the paths leads to the end vertex, then we do not need to push the start vertex as it is neither directly nor indirectly connected to the end vertex, hence we will simply return NULL. In case any of the neighbors return a non-null entry, it means that we have a path from that neighbor to the end vertex, hence we can now insert the start vertex into the solution array.

Try to code it yourself, and for the answer, refer to the solution tab of the same.

Get Path - BFS

Approach: It is the same problem as the above, just we have to code the same using BFS.

Approach: Using BFS will provide us the shortest path between the two vertices. We will use the queue over here and do the same until the end vertex gets inserted into the queue. Here, the problem is how to figure out the node, which led us to the end vertex. To overcome this, we will be using a map. In the map, we will store the resultant node as the index, and its key will be the node that led it into the queue.

For example: If the graph was such that 0 was connected to 1 and 0 was connected to 2, and currently, we are on node 0 such that node 1 and node 2 are not visited. So our map will look as follows:

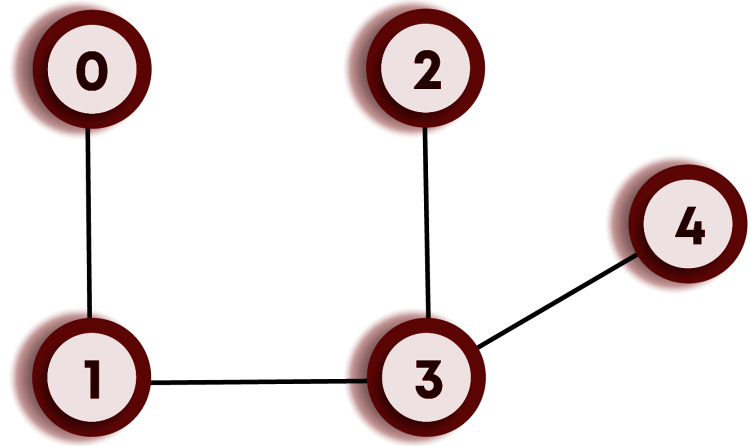
1	0
2	0

This way, as soon as we reach the end vertex, we can figure out the nodes by running the loop until we reach the start vertex as the key value of any node.

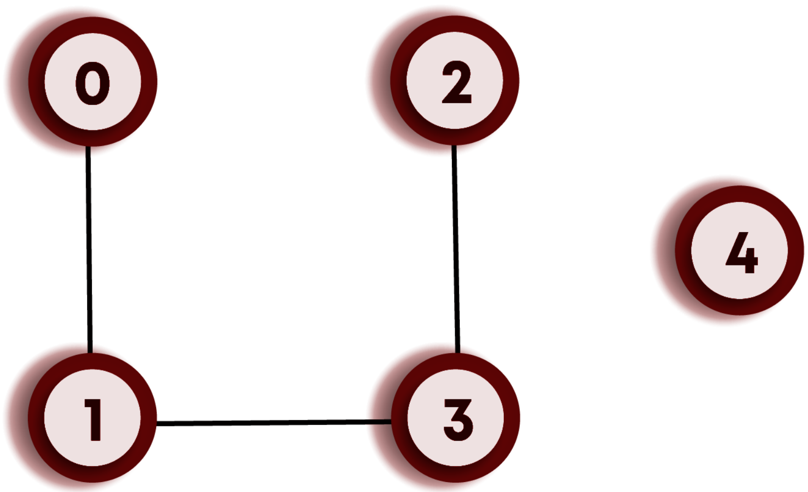
Try to code it yourselves, and for the solution, refer to the specific tab of the same.

Is connected?

Problem statement: Given an undirected graph G(V, E), check if the graph G is a connected graph or not. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.



Connected graph



Disconnected graph

Example 1: Suppose the given input is:

```
4 4
0 1
0 3
1 2
2 3
1 3
```

The output should be: **true**

Explanation: As the graph is connected, so according to the question, the answer will be true.

Example 2: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be: **false**

Explanation: The graph is not connected, even though vertices 0,1, and 3 are connected, but there isn't any path from vertices 0,1,3 to vertex 2. Hence, according to the question, the answer will be false.

Approach: This is very start-forward. Take any vertex as the starting vertex as traverse the graph using either DFS or BFS. In the end, check if all the vertices are visited or not. If not, it means that the node was not connected to the starting vertex, which means it is a disconnected graph. Otherwise, it is a connected graph. Try to code it yourselves, and for the code, refer to the solution tab of the same.

Problem statement: Return all Connected Components

Given an undirected graph G(V, E), find and print all the connected components of the given graph G. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

You need to take input in the main and create a function that should return all the connected components. And then print them in the main, not inside a function.

Print different components in a new line. And each component should be printed in increasing order (separated by space). The order of different components doesn't matter.

Example: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be:

```
0 1 3
2
```

Explanation: As we can see that {0, 1, 3} is one connected component, and {2} is the other one. So, according to the question, we just have to print the same.

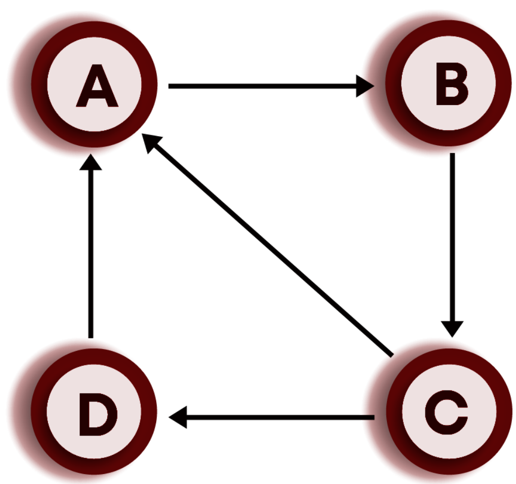
Approach: For this problem, start from vertex 0 and traverse until vertex n-1. If the vertex is not visited, then run DFS/BFS on it and keep track of all the connected vertices through that node. This way, we will get all the distinct connected components, and we can print them at last.

This problem is left for you to solve. For the code, refer to the solution tab of the same.

Weighted and Directed Graphs

There are two more variations of the graphs:

Directed graphs: These are generally required when we have one-way routes. Suppose you can go from node A to node B, but you cannot go from node B to node A. Another example could be social media(like Twitter) if you are following someone, it does not mean that they are following you too.



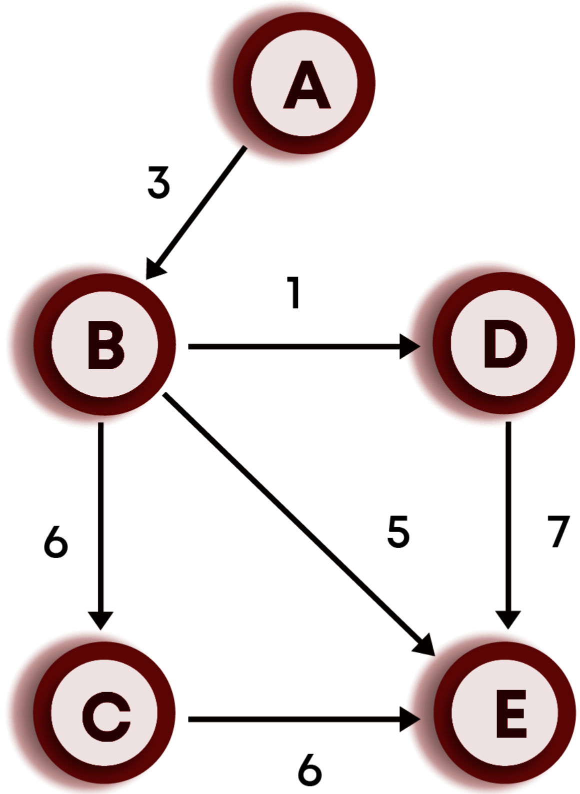
To implement these, there is a small change in the implementation of indirect graphs. In indirect graphs, if there was an edge between node i and j, then we did:

```
edges[i][j] = 1;
edges[j][i] = 1;
```

But, in the case of a directed graph, we will just do the following:

```
edges[i][j] = 1;
```

Weighted graphs: These generally mean that all the edges are not equal, which means somehow, each edge has some weight assigned to it. This weight can be the length of the road connecting the cities or many more.



To implement this, in the edges matrix, we will assign a weight to connected nodes instead of putting it 1 at that position. For example: If node i and j are connected, and the weight of the edge connecting them is 5, then `edges[i][j] = 5`.

Practice problems:

- https://www.codechef.com/problems/CHEFDAG
- https://www.spoj.com/problems/WORDS1/
- https://www.hackerrank.com/challenges/the-quickest-way-up/problem