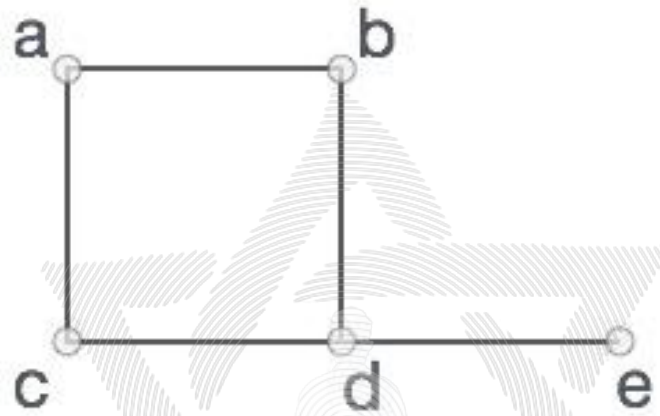
Graphs
Lesson 1

1. depth-first-traversal-for-a-graph
2. print-adjacency-list
3. bfs-traversal-of-graph

Graph Data Structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

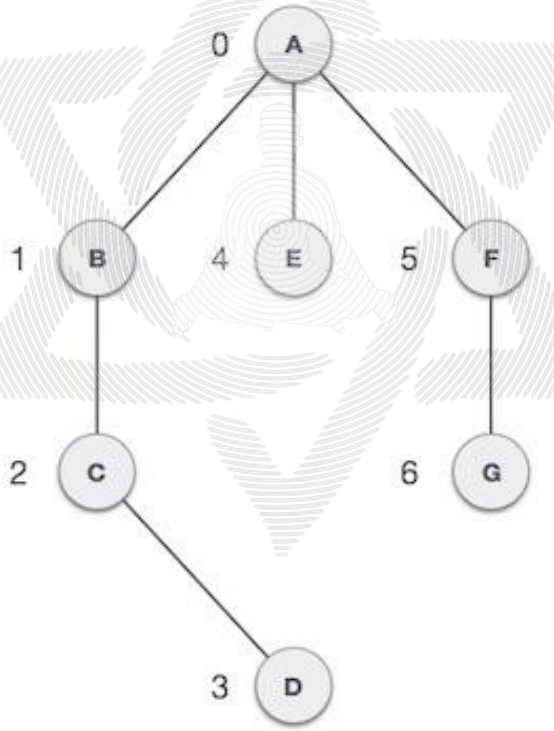
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- Vertex − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

- Edge − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represent edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- Adjacency − Two nodes or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

- Path − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Directed graph
Undirected graph

A graph is said to be directed if the edges of the graph have directions. In directed graphs, there exists a one way relationship between two nodes, and the edge can be traversed in a single direction only. A graph is said to be undirected if the edges of the graph have no directions and there exists a two-way relationship between two nodes, and the edge can be traversed in both directions.

Here's an example of a directed graph. See how node F can be traversed from node E but the vice versa cannot, as edge (E, F) is directed only from E to F.

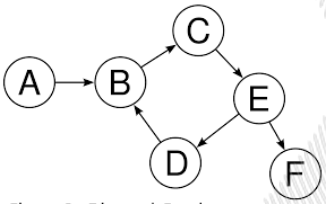


Figure 3 : Directed Graphs

In this example of undirected graph, node D can be traversed from node A by path {(A, B), (B, D)} and node A can be traversed from node D by path {(D, B), (B, A)}. The edges are not directed in a single direction, and thus, can be traversed in both directions.

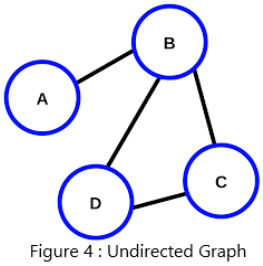


Figure 4 : Undirected Graph

Cyclic and acyclic graph

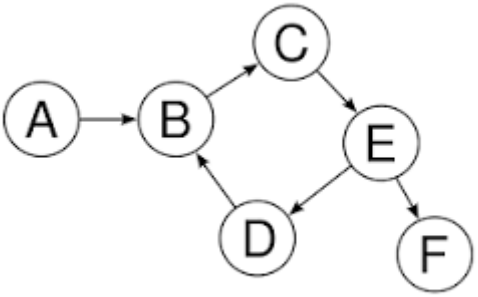
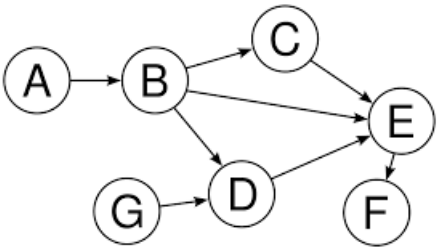
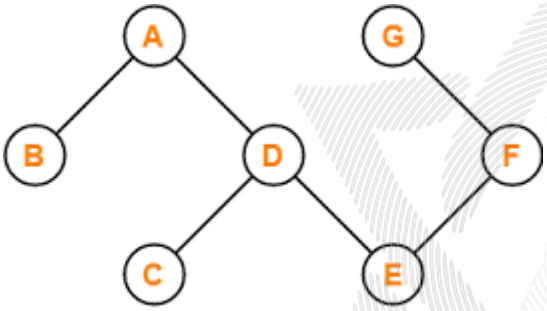
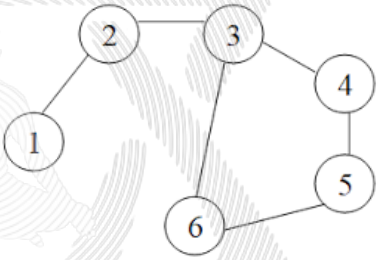


Figure 5 : Cyclic Graph





Example of Acyclic Graph



Graph and its representations

Adjacency Matrix:

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 an undirected graph is always symmetric. The 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.

Adjacency List:

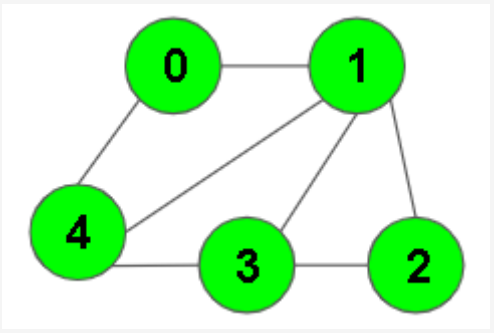
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

Print adjacency list

Given the adjacency list of a bidirectional graph. Your task is to return the adjacency list for each vertex.

Example 1:

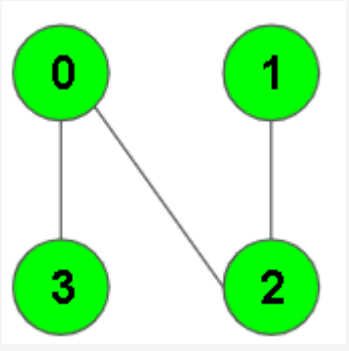
Input:



Output:
0-> 1-> 4
1-> 0-> 2-> 3-> 4
2-> 1-> 3
3-> 1-> 2-> 4
4-> 0-> 1-> 3

Example 2:

Input:



Output:
0-> 2-> 3
1-> 2
2-> 0-> 1
3-> 0

Expected Time Complexity: O(V + E)

Expected Auxiliary Space: O(1)

Constraints:

1 ≤ V, E ≤ 104

```
#include<bits/stdc++.h>
using namespace std;
class Solution
{
  public:
      //Function to return the adjacency list for each vertex.
      vector<vector<int>>printGraph(int V, vector<int> adj[])
      {
         vector<vector<int>>v1;
         for(int i=0;i<V;i++)
         {
            vector<int>v2;
            v2.push_back(i);
            for(int j=0;j<adj[i].size();j++)
            {
               v2.push_back(adj[i][j]);
            }
            v1.push_back(v2);
         }
         return v1;
            }
};


int main(){
      int tc;
```

```
27.         cin >> tc;
28.         while(tc--){
29.                 int V, E;
30.                 cin >> V >> E;
31.                 vector<int>adj[V];
32.                 for(int i = 0; i < E; i++){
33.                         int u, v;
34.                         cin >> u >> v;
35.                         adj[u].push_back(v);
36.                         adj[v].push_back(u);
37.                 }
38.                 Solution obj;
39.                 vector<vector<int>>ans=obj.printGraph(V, adj);
40.         for(int i=0;i<ans.size();i++){
41.         for(int j=0;j<ans[i].size()-1;j++){
42.                 cout<<ans[i][j]<<"-> ";
43.         }
44.         cout<<ans[i][ans[i].size()-1];
45.         cout<<endl;
46.         }
47.         }
48.         return 0;
49.   }  // } Driver Code Ends
```

Depth First Search or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, and a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

Example:

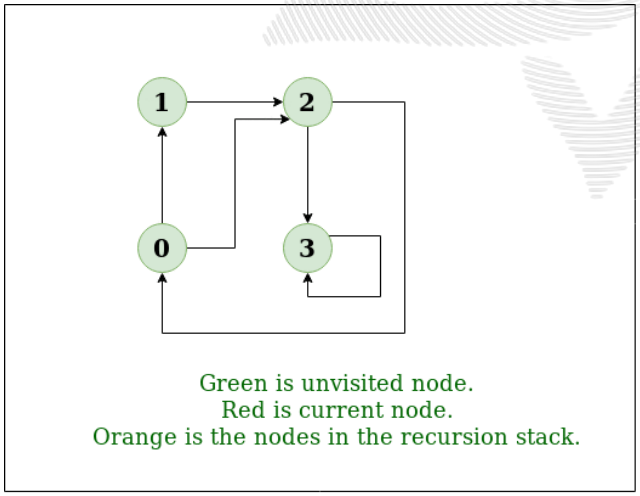
Input: n = 4, e = 6

0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3

Output: DFS from vertex 1 : 1 2 0 3

Explanation:

DFS Diagram:



Solution:

1. Approach: Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.
2. Algorithm:
    1. Create a recursive function that takes the index of the node and a visited array.
    2. Mark the current node as visited and print the node.
    3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

```
1.    void DFS(int x , vector<int> adj[] , vector<int>&v , bool visited[])
2.      {
3.         if(visited[x]==false)
4.         {
5.            v.push_back(x);
6.            visited[x]=true;
7.         }
8.         for(int i=0;i<adj[x].size();i++)
9.         {
10.           if(visited[adj[x][i]]==false)
11.              DFS(adj[x][i] , adj ,v , visited);
12.        }
13.     }
14.        //Function to return a list containing the DFS traversal of the graph.
15.        vector<int>dfsOfGraph(int V, vector<int> adj[])
16.        {
17.           vector<int>v;
18.           bool visited[10001]={false};
19.           DFS(0,adj, v , visited);
20.           return v;
21.           // Code here
22.        }
```

Complexity Analysis:

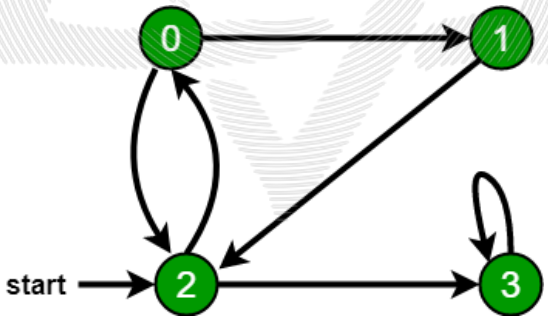
- Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- Space Complexity: O(V).

  Since an extra visited array is needed of size V.

Breadth First Search or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. The Breadth First Traversal of the following graph is 2, 0, 3, 1.



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

// } Driver Code Ends


class Solution
{
  public:
  //Function to return Breadth First Traversal of given graph.
      vector<int>bfsOfGraph(int V, vector<int> adj[])
      {
         vector<int>v;
         bool visited[V]={false};
         queue<int>q;
         q.push(0);
         visited[q.front()]=true;
         while(!q.empty())
         {
           v.push_back(q.front());

        for(int i=0;i<adj[q.front()].size();i++)
        {
           if(visited[adj[q.front()][i]] == false)
           {
              q.push(adj[q.front()][i]);
              visited[adj[q.front()][i]]=true;
           }
        }
        q.pop();
         }
```

```
33.             return v;
34.             // Code here
35.         }
36. };
37. 
38. // { Driver Code Starts.
39. int main(){
40.         int tc;
41.         cin >> tc;
42.         while(tc--){
43.                 int V, E;
44.         cin >> V >> E;
45. 
46.         vector<int> adj[V];
47. 
48.         for(int i = 0; i < E; i++)
49.         {
50.                 int u, v;
51.                 cin >> u >> v;
52.                 adj[u].push_back(v);
53.     //          adj[v].push_back(u);
54.         }
55.             Solution obj;
56.         vector<int>ans=obj.bfsOfGraph(V, adj);
57.         for(int i=0;i<ans.size();i++){
58.          cout<<ans[i]<<" ";
59.         }
60.         cout<<endl;
61.          }
62.         return 0;
63. }  // } Driver Code Ends
```