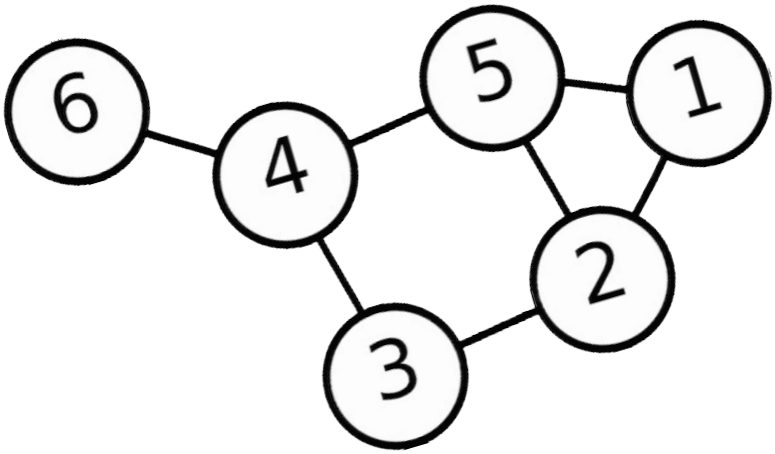
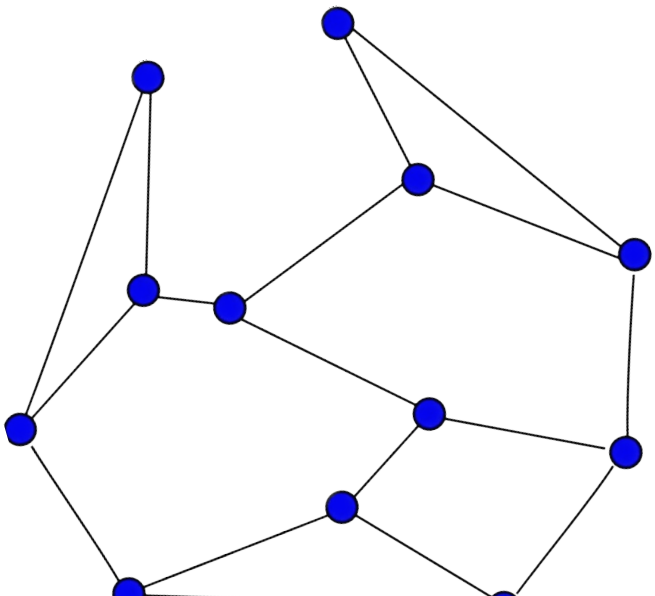


Graphs (Part-1)

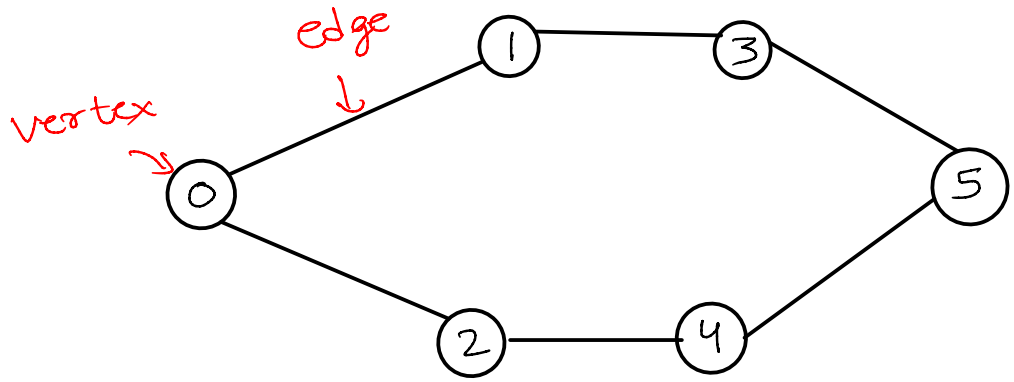
- By Kapil Yadav

1. BFS using adjacency matrix
2. BFS using adjacency list
3. DFS using adjacency matrix
4. DFS using adjacency list



Graph

A Graph G(V,E) is a set of Vertices and Edges.



$V = \{0, 1, 2, 3, 4, 5\}$

$E = \{\{0,1\}\ \{0,2\}\ \{1,3\}\ \{2,4\}\ \{3,5\}\ \{4,5\}\}$

→ Two vertices are said to be adjacent, if there is an edge between them.

Order of Graph:-

→ The number of vertices in a graph represents order of graph.

$$O(G) = 6$$

Size of Graph:-

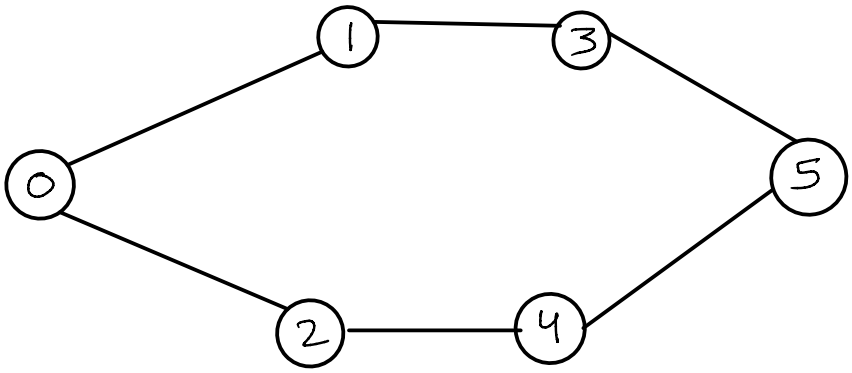
→ No. of edges in a graph

$$Size(G) = e = 6$$

Types of Graph:-

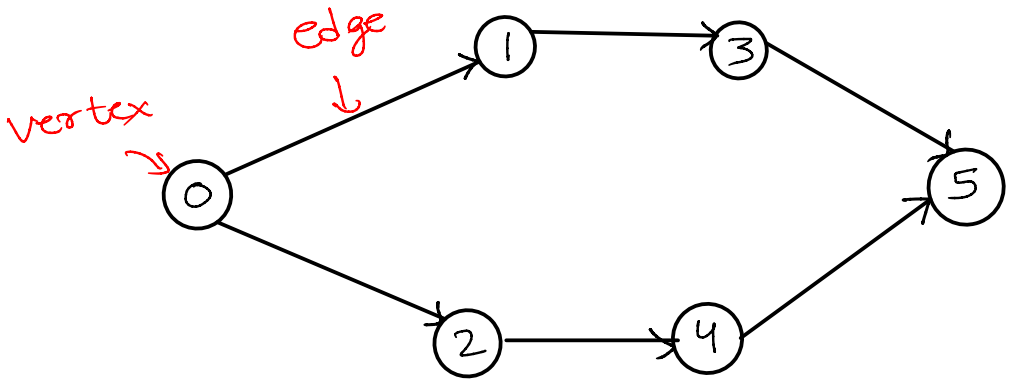
- Graph
  - Undirected Graph
    - Simple Graph
    - Multi Graph
    - Pseudo - Graph
  - Directed graph
    - Simple Directed Graph
    - Multi Graph
    - Pseudo Graph

1) Undirected Graph:- Edges in Undirected graph are Bidirectional.



Undirected graph

2. Directed Graph:- Graph having directed edges.

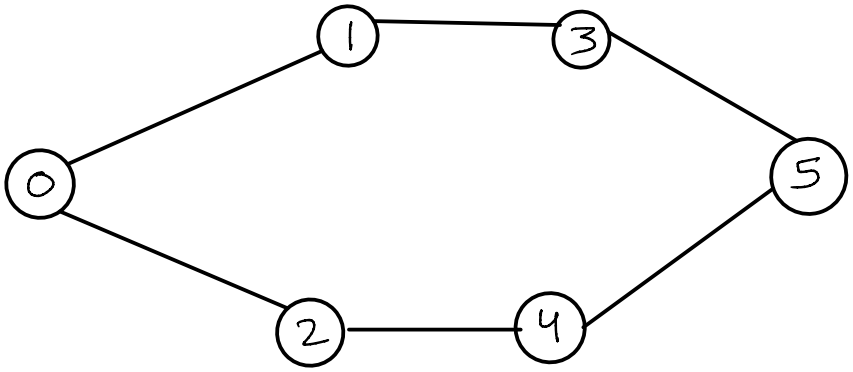


Simple Graph :-
- → No self loops.
- → No Multiple edges.

Multi Graph :-
- → Multiple edges are allowed.
- → Self loops are not allowed.

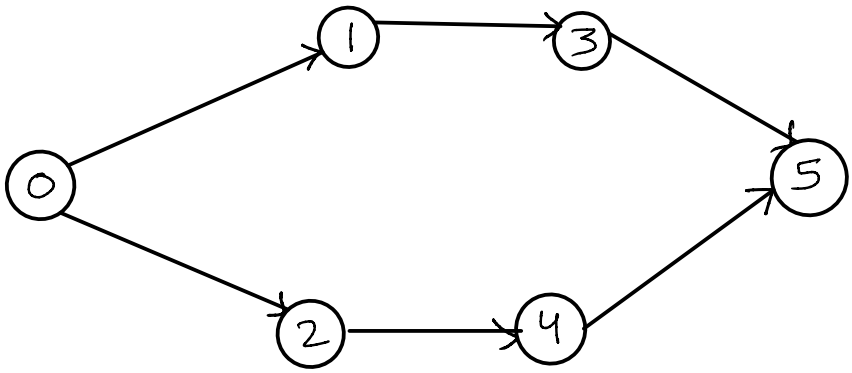
pseudo Graph :-
- → Self loops are allowed
- → Multiple edges are allowed.

Degree of a vertex :-
- → Number of vertices adjacent to a vertex V.



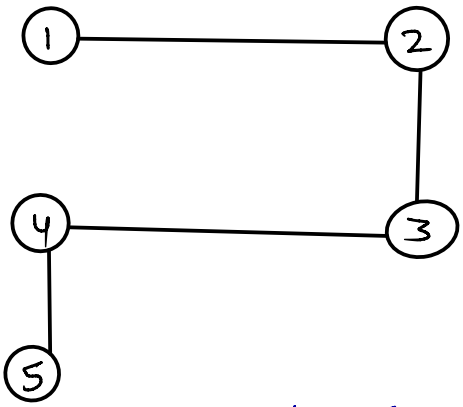
$deg(2) = 2$

→ Incase of Directed graph, each vertex has an indegree and an outdegree.



Indegree (2) = 1
Outdegree (2) = 1

Path :- Path is a sequence of alternating vertices and edges such that the edge connects each successive vertex.



Path = 1 2 3 4 5

→ Edge and Vertex should not be repeat in a path.

Cycle :- Cycle is a path that starts and ends at the same vertex.

1 2 3 4 1
3 4 1 2 3
⋮
4 1 2 3 4

→ Tree is a connected graph with no cycles.

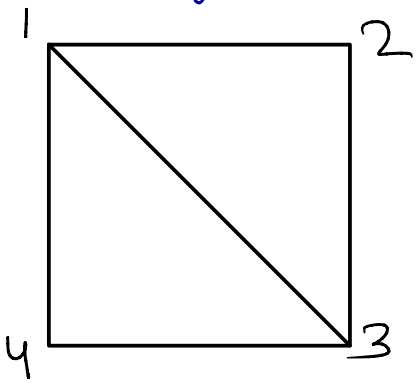
Graph Representation :-

1. Adjacency Matrix
2. Adjacency list
3. Incidence Matrix (when edges are labelled)

1. Adjacency Matrix :- A graph G with V vertices, can be represent using V × V matrix

$$\begin{bmatrix} a_{11} & a_{12} \cdots a_{1n} \\ a_{21} & a_{22} \cdots a_{2n} \\ \vdots & \\ a_{n1} & a_{n2} \cdots a_{nn} \end{bmatrix}$$

→ In Undirected graph, the value $a_{ij} = a_{ji}$, and the adjacency matrix is symmetric.



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

SC = O(V × V)

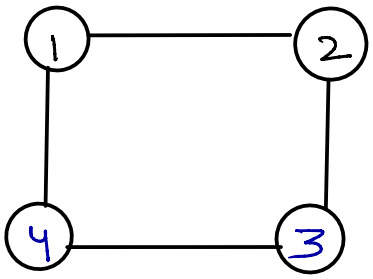
```cpp
//Adjacency Matrix
int main() {
    int V,E;
    cin>>V;
    cin>>E;
     //graph having Vertices from 0 to n-1
     //if vertices are from 1 to n,take n+1,n+1 size adjamtrix
    vector<vector<int>> adjmat(V, vector<int> (V,0));
    int i=0;

    while(i<E){
        int from,to;
        cin>>from>>to;
        adjmat[from][to] = 1;
        adjmat[to][from] = 1;
        i++;
    }
```

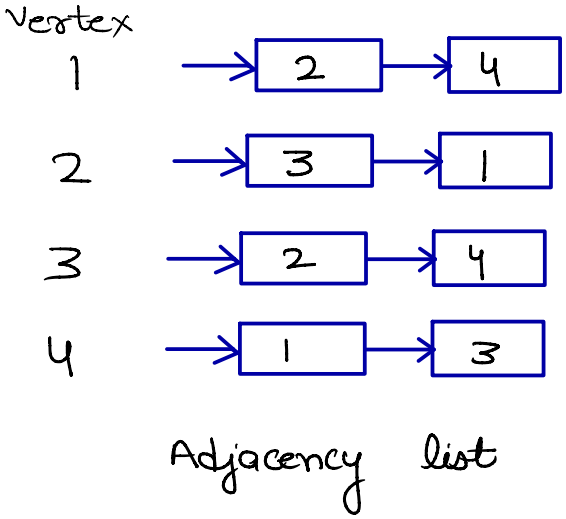
Adjacency Matrix of Undirected Graph

→ for Directed graph, remove line 15.

2. Adjacency list :-

→ We use an array of vector to represent the graph.





Adjacency list

Space = O(2E)

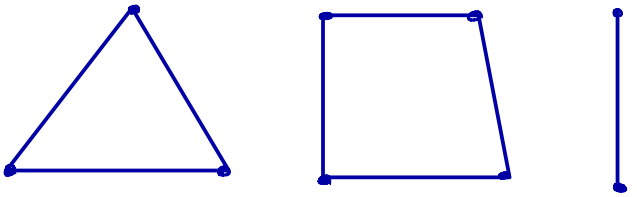
```cpp
//Adjacency List
int main() {
    int V,E;
    cin>>V;
    cin>>E;
     //graph having Vertices from 0 to n-1
   vector<int> adjlist[V];
    int i=0;
    while(i<E){
         int from,to;
        cin>>from>>to;
        adjlist[from].push_back(to);
        adjlist[to].push_back(from);
        i++;
    }
```

Adjacency list of Undirected Graph

→ for Directed graph, remove line 13.

→ Space Complexity for Directed graph = O(E).

Connected Components :- No. of connected subgraph in a given graph.



→ Number of components in connected graph = 1

Breadth-First Search : BFS is one of the traversal techniques.

→ There can be multiple connected components, so we need to have visited array.

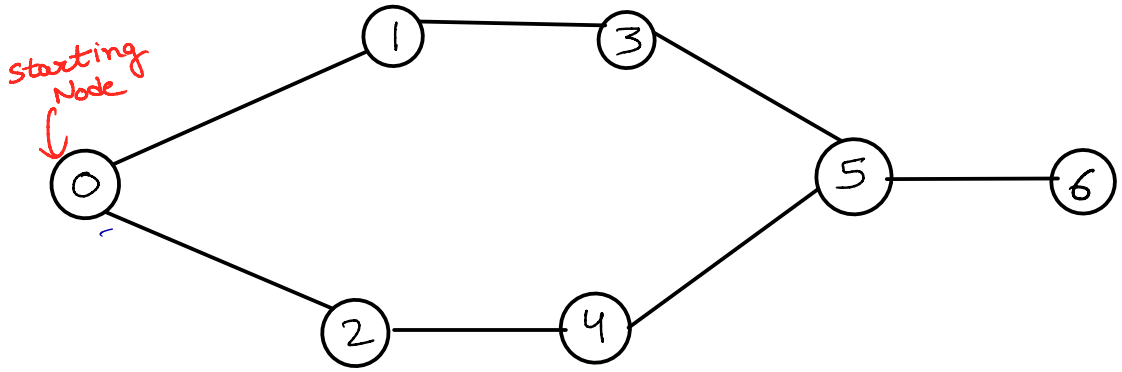
→ Each node will be either visited or not visited.

→ BFS traverse the graph layerwise, so we use Queue data structure for traversal.

Algorithm :-

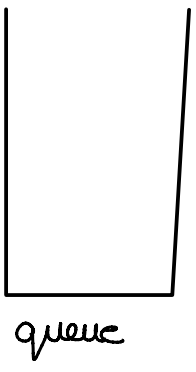
- Take a queue and insert the starting node.
- Declare a visited array and initialize it with 0.
- Mark the starting vertex as visited.
- Follow the steps, until the queue become empty.
  - → Remove front vertex from queue.
  - → Insert all the unvisited neighbours of the vertex into the queue and mark them visited.

Example 1:



visited

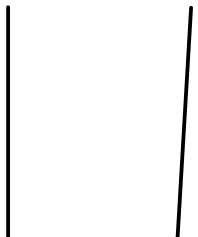
0	1	2	3	4	5	6
0	0	0	0	0	0	0



1. Push 0 in queue

visited

0	1	2	3	4	5	6
1	0	0	0	0	0	0

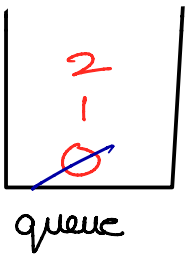


2. While queue does not get empty,

Pop 0, Push 1, Push 2

visited

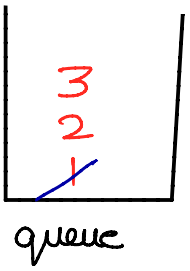
0	1	2	3	4	5	6
1	1	1	0	0	0	0



3.) Pop 1, Push 3

visited

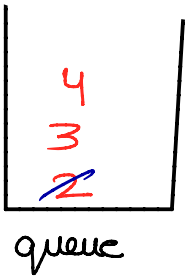
0	1	2	3	4	5	6
1	1	1	1	0	0	0



4.) Pop 2, Push 4

visited

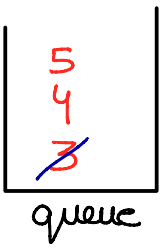
0	1	2	3	4	5	6
1	1	1	1	1	0	0



5.) Pop 3, Push 5

visited

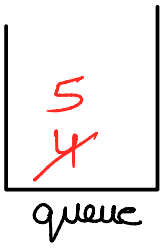
0	1	2	3	4	5	6
1	1	1	1	1	1	0



6.) Pop 4

visited

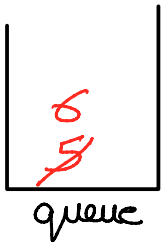
0	1	2	3	4	5	6
1	1	1	1	1	1	0



7.) Pop 5, Push 6

visited

0	1	2	3	4	5	6
1	1	1	1	1	1	1



ans

8.) Pop 6

{0,1,2,3,

BFS using Adjacency Matrix (Undirected Graph)

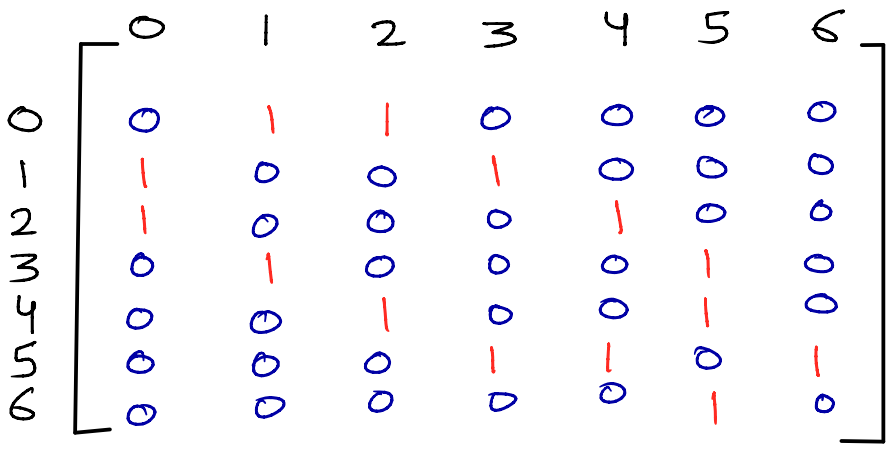
```cpp
#include <iostream>
#include<vector>
#include<queue>
using namespace std;

//BFS using Adjacency Matrix
vector<int> bfs(vector<vector<int>>& adjmat){
  int V = adjmat.size();
 vector<int> vis(adjmat.size(),0);
 vector<int> ans;
 queue<int> q;
 q.push(0);
 vis[0] = 1;
 while(!q.empty()){
     int currele = q.front();
     q.pop();
     ans.push_back(currele);
     for(int i=0;i<V;i++){
         if(vis[i]==0 && adjmat[currele][i]==1){
             vis[i]=1;
             q.push(i);
         }
     }
 }
return ans;
}

void printmatrix(vector<vector<int>>& adjmat){
    int V = adjmat.size();
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            cout<<adjmat[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

$TC = O(V \times V)$

$SC = O(V \times V) + O(V)$

↓ Adj matrix ↓ to store answer

```cpp
int main() {
    int V,E;
    cin>>V;
    cin>>E;
     //graph having Vertices from 0 to n-1
     //if vertices are from 1 to n,take n+1,n+1 size adjamtrix
    vector<vector<int>> adjmat(V, vector<int> (V,0));
    int i=0;

    while(i<E){
         int from,to;
        cin>>from>>to;
        adjmat[from][to] = 1;
        adjmat[to][from] = 1;
        i++;
    }
//printmatrix(adjmat);
vector<int> ans = bfs(adjmat);

for(int i=0;i<ans.size();i++){
    cout<<ans[i]<<" ";
}

return 0;
}
```



BFS Using Adjacency List (Undirected Graph)

```cpp
#include <iostream>
#include<vector>
using namespace std;

 vector<int> bfs(int V, vector<int> adj[]) {
        vector<int> ans;
        vector<bool> vis(V,0);
        queue<int> q;
        q.push(0);
        vis[0] = 1;

        while(!q.empty()){
            int currele = q.front();
            ans.push_back(currele);
            q.pop();
            for(int i=0;i<adj[currele].size();i++){
                if(!vis[adj[currele][i]]){
                    vis[adj[currele][i]] =1;
                    q.push(adj[currele][i]);
                }
            }
        }
        return ans;
    }

void printlist(vector<int> adjlist[],int V){
    for(int i=0;i<V;i++){
        cout<<i<<"->  ";
        for(int j=0;j<adjlist[i].size();j++){
            cout<<adjlist[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

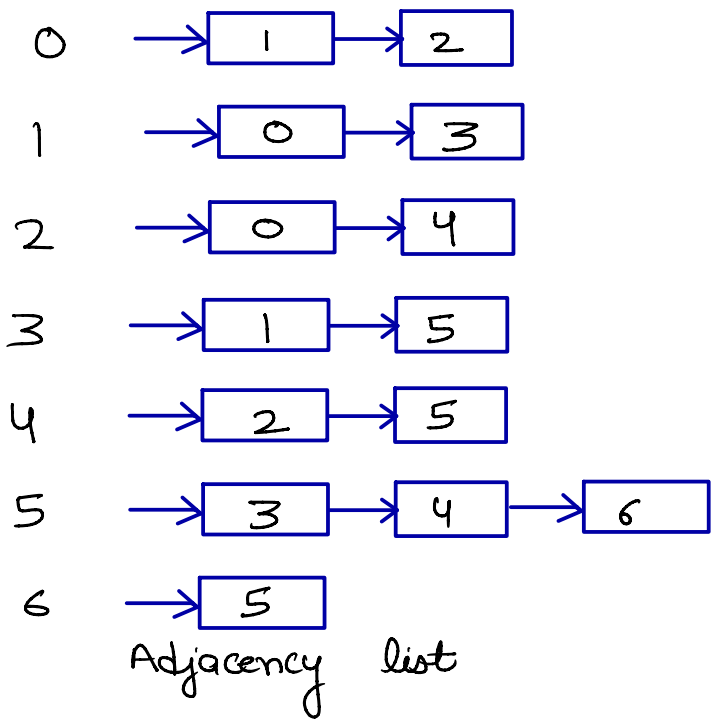
$$TC = O(V) + O(2E)$$

$O(V)$: No. of vertices

$O(2E)$: total degrees

$$SC = O(V+E)$$
$$= O(V^2) \quad \{\text{In Worst Case}\}$$

```cpp
int main() {
    int V,E;
    cin>>V;
    cin>>E;
     //graph having Vertices from 0 to n-1
   vector<int> adjlist[V];
    int i=0;
    while(i<E){
         int from,to;
        cin>>from>>to;
        adjlist[from].push_back(to);
        adjlist[to].push_back(from);
        i++;
    }
printlist(adjlist,V);
  vector<int> ans = bfs(V,adjlist);

cout<<"BFS USING ADJACENCY LIST  : ";
for(int i=0;i<ans.size();i++){
    cout<<ans[i]<<" ";
}

return 0;
}
```



Depth First Search (DFS):

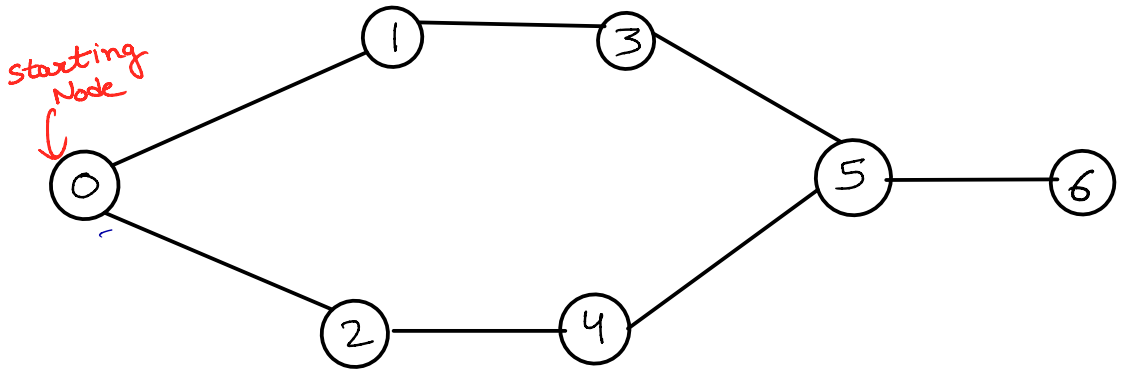
Depth first search is a recursive algorithm.

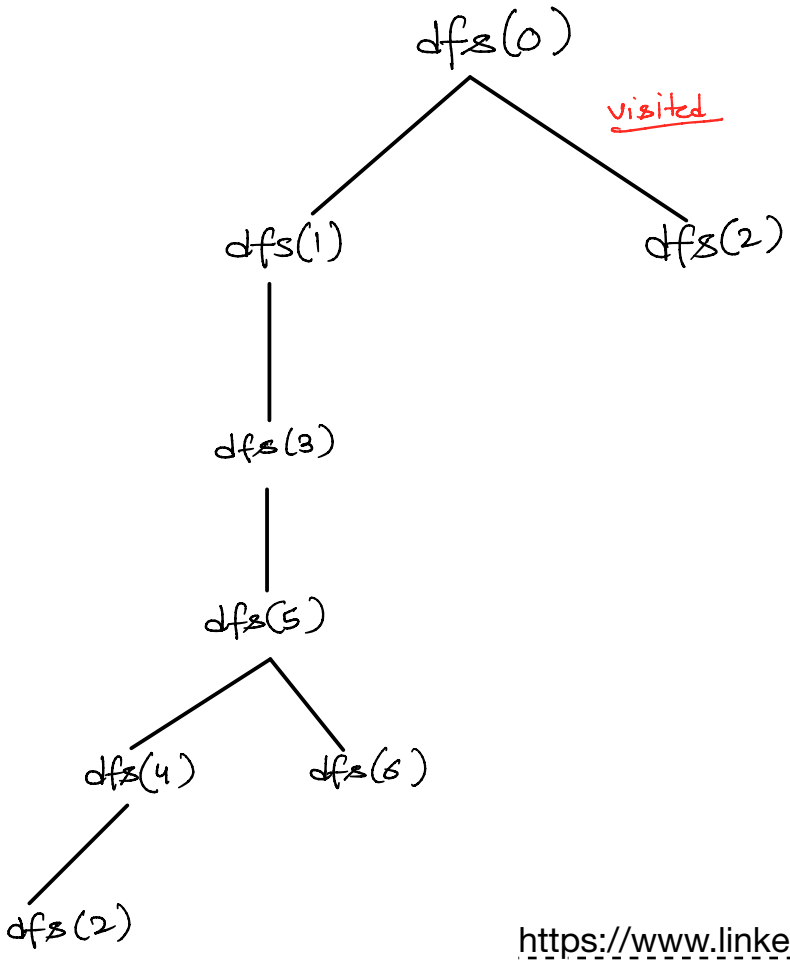
→ In DFS, we select a starting node, and explore it as far as possible.

Algorithm :-

- Choose a starting node and do recursion.
- Make current node as visited and add current node in our output list (vector/array).
- Traverse all the adjacent and unvisited nodes.

Ex -





→ We can choose neighbour nodes in any order. But each node will traverse only one.

0 2 4 5 6 3 1

0 2 4 5 3 1 6

```cpp
//DFS Using Adjacency Matrix
void dfs(int currver,vector<vector<int>>& adjmat, vector<bool>& vis,vector<int>& ans){
    ans.push_back(currver);
    vis[currver]=1;
    int V = adjmat.size();
    for(int i=0;i<V;i++){
        if(!vis[i] && adjmat[currver][i]==1){
            dfs(i,adjmat,vis,ans);
        }
    }
}

int main() {
    int V,E;
    cin>>V;
    cin>>E;
     //graph having Vertices from 0 to n-1
     //if vertices are from 1 to n,take n+1,n+1 size adjamtrix
    vector<vector<int>> adjmat(V, vector<int> (V,0));
    int i=0;

    while(i<E){
         int from,to;
        cin>>from>>to;
        adjmat[from][to] = 1;
        adjmat[to][from] = 1;
        i++;
    }

    vector<int> ans;
    vector<bool> vis(V,0);
    dfs(0,adjmat,vis,ans);
    cout<<" DFS USING ADJACENCY MATRIX  : ";
    for(int i=0;i<ans.size();i++){
    cout<<ans[i]<<" ";
    }
return 0;
}
```

```cpp
//DFS Using Adjacency list

void dfs(int currver,vector<int> adjlist[], vector<bool>& vis,vector<int>& ans){
    ans.push_back(currver);
    vis[currver]=1;
    for(auto i : adjlist[currver]){
        if(!vis[i]){
            dfs(i,adjlist,vis,ans);
        }
    }
}

int main() {
    int V,E;
    cin>>V;
    cin>>E;
     //graph having Vertices from 0 to n-1
   vector<int> adjlist[V];
    int i=0;
    while(i<E){
         int from,to;
        cin>>from>>to;
        adjlist[from].push_back(to);
        adjlist[to].push_back(from);
        i++;
    }
    vector<bool> vis(V,0);
vector<int> ans;
dfs(0,adjlist,vis,ans);

for(int i=0;i<ans.size();i++){
    cout<<ans[i]<<" ";
}
return 0;
}
```