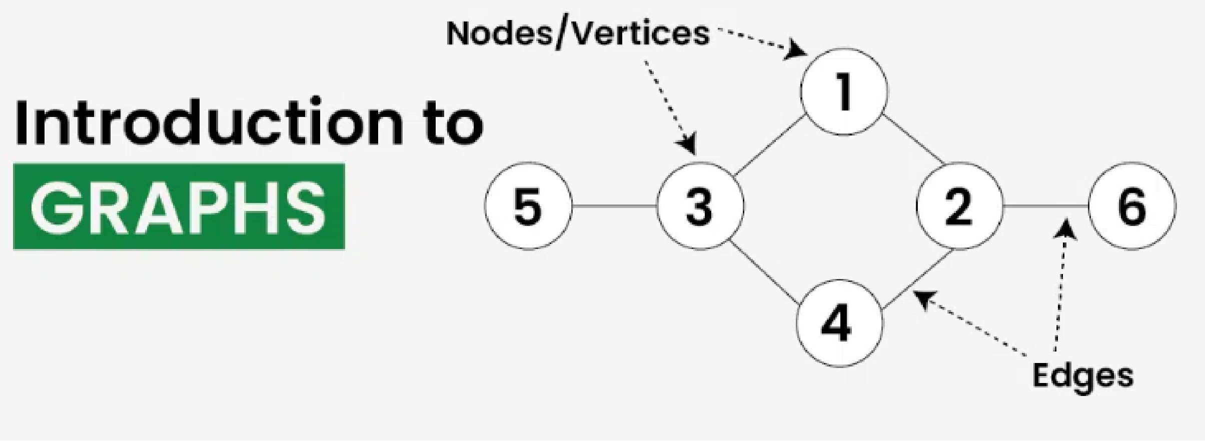
Graphs

What is a graph?

A graph is a non linear data structure. Graph consists of a finite set of vertices(or nodes) and a set of vertices which connect a pair of nodes.



Components of graph data structure

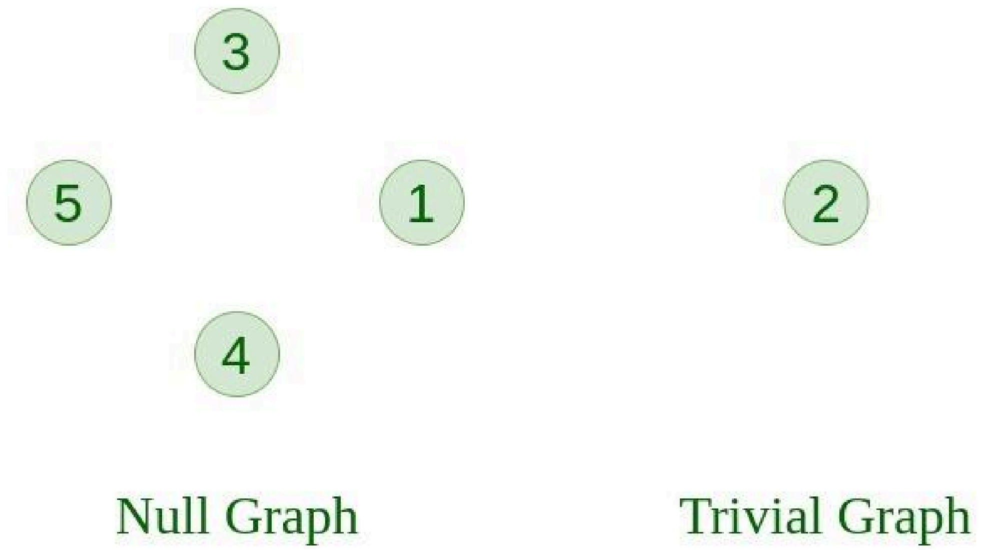
1. **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

2. **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

Types of graph

1. Null Graph: A graph is known as a null graph if there are no edges in the graph.

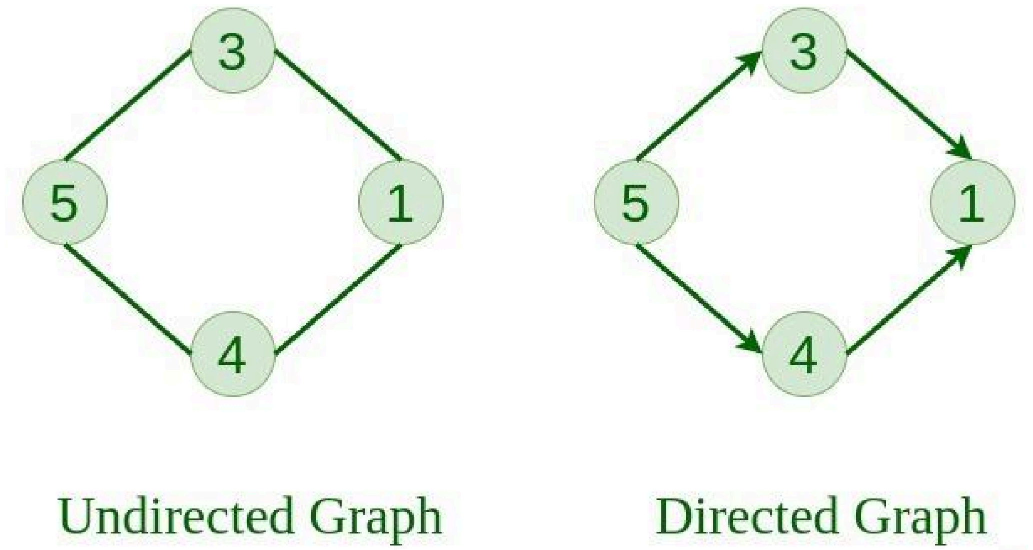
2. Trivial Graph: Graph having only a single vertex, it is also the smallest graph possible.



Null Graph Trivial Graph

3. Undirected Graph: A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

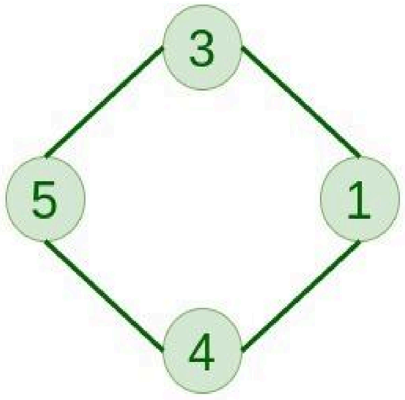
4. Directed Graph: A graph in which the edge has direction. That is the nodes are ordered pairs in the definition of every edge.



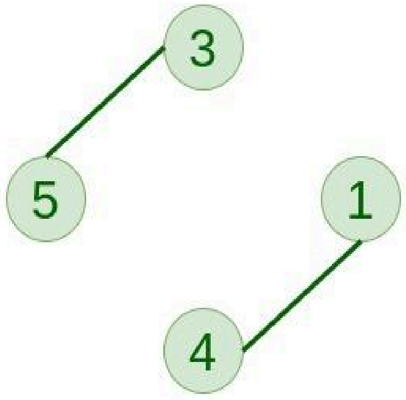
Undirected Graph Directed Graph

5. Connected Graph: The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph: The graph in which at least one node is not reachable from a node is known as a disconnected graph.

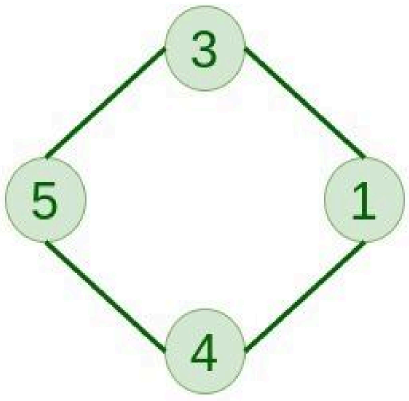


Connected Graph

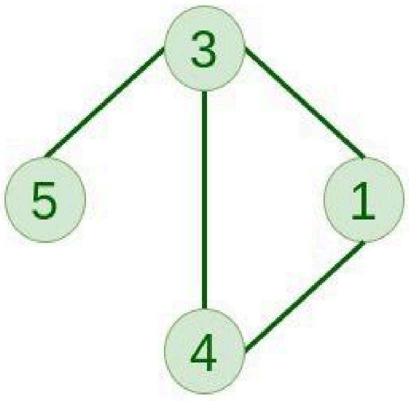


Disconnected Graph

7. Regular Graph: The graph in which the degree of every vertex is equal to K is called K regular graph.
8. Complete Graph: The graph in which from each node there is an edge to each other node.
9. Cycle Graph: The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.
10. Cyclic Graph: A graph containing at least one cycle is known as a Cyclic graph.

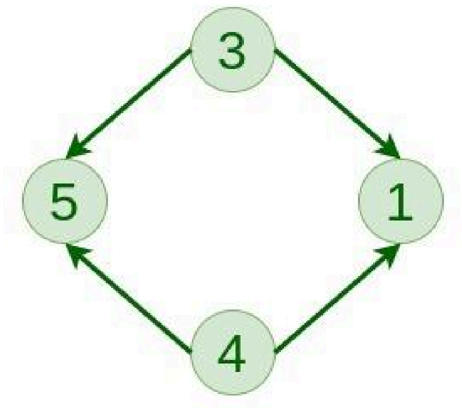


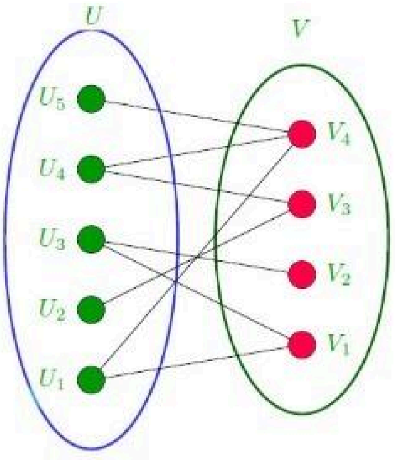
Cycle Graph



Cyclic Graph

11. Directed Acyclic Graph: A Directed Graph that does not contain any cycle.
12. Bipartite Graph: A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.





Directed Acyclic Graph Bipartite Graph

13. Weighted Graph:

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

Representation of graph

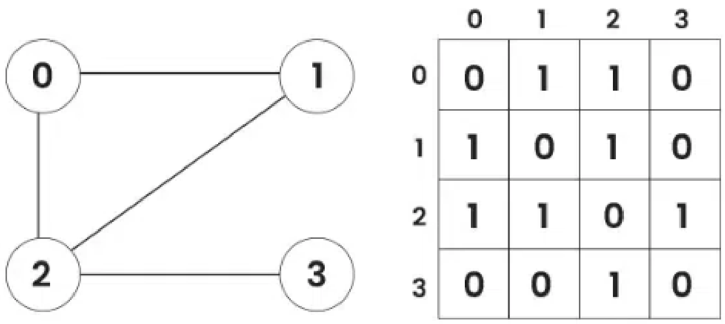
There are multiple ways to store a graph: The following are the most common representations.

- Adjacency Matrix
- Adjacency List

Adjacency matrix

Graph is stored in the form of 2D matrix.
Rows and columns represent vertices.
Each entry in the matrix represents the weight of the edge between those vertices.

Adjacency Matrix of Graph



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

Code in python:

```python
def add_edge(mat, i, j):

    # Add an edge between two vertices
    mat[i][j] = 1  # Graph is
    mat[j][i] = 1  # Undirected

def display_matrix(mat):

    # Display the adjacency matrix
    for row in mat:
        print(" ".join(map(str, row)))

# Main function to run the program
if __name__ == "__main__":
    V = 4  # Number of vertices
    mat = [[0] * V for _ in range(V)]

    # Add edges to the graph
    add_edge(mat, 0, 1)
    add_edge(mat, 0, 2)
    add_edge(mat, 1, 2)
    add_edge(mat, 2, 3)

    # Optionally, initialize matrix directly
    """
    mat = [
        [0, 1, 0, 0],
        [1, 0, 1, 0],
        [0, 1, 0, 1],
        [0, 0, 1, 0]
    ]
    """

    # Display adjacency matrix
    print("Adjacency Matrix:")
    display_matrix(mat)
```

Code in c++:

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

void addEdge(vector<vector<int>> &mat, int i, int j)
{
    mat[i][j] = 1;
    mat[j][i] = 1; // Since the graph is undirected
}

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

int main()
{

    // Create a graph with 4 vertices and no edges
    // Note that all values are initialized as 0
    int V = 4;
    vector<vector<int>> mat(V, vector<int>(V, 0));

    // Now add edges one by one
    addEdge(mat, 0, 1);
    addEdge(mat, 0, 2);
    addEdge(mat, 1, 2);
    addEdge(mat, 2, 3);

    /* Alternatively we can also create using below
       code if we know all edges in advacem

     vector<vector<int>> mat = {{ 0, 1, 0, 0 },
                                { 1, 0, 1, 0 },
                                { 0, 1, 0, 1 },
                                { 0, 0, 1, 0 } }; */

    cout << "Adjacency Matrix Representation" << endl;
    displayMatrix(mat);

    return 0;
}
```

Code in javascript:

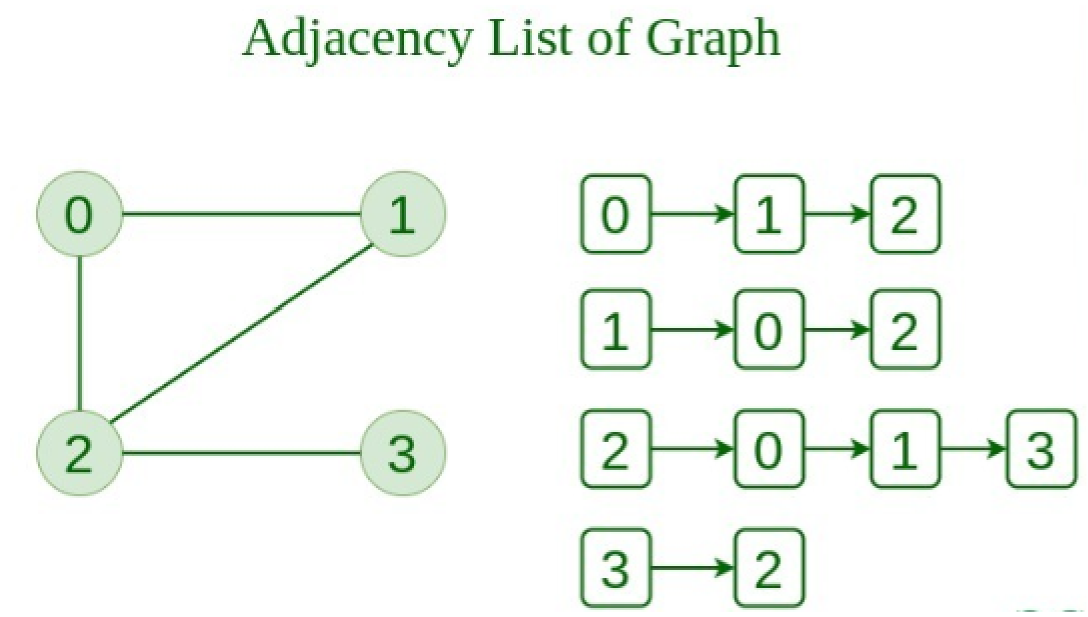
```
function addEdge(mat, i, j) {
    mat[i][j] = 1; // Graph is
    mat[j][i] = 1; // undirected
}

function displayMatrix(mat) {
    // Display the adjacency matrix
    for (const row of mat) {
        console.log(row.join(" "));
    }
}

// Main function to run the program
const V = 4; // Number of vertices

 // Initialize matrix
let mat = Array.from({ length: V }, () => Array(V).fill(0));

// Add edges to the graph
addEdge(mat, 0, 1);
addEdge(mat, 0, 2);
addEdge(mat, 1, 2);
addEdge(mat, 2, 3);

/* Optionally, initialize matrix directly
let mat = [
    [0, 1, 0, 0],
    [1, 0, 1, 0],
    [0, 1, 0, 1],
    [0, 0, 1, 0]
];
*/

// Display adjacency matrix
console.log("Adjacency Matrix:");
displayMatrix(mat);
```

Output

```
Adjacency Matrix Representation
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
```

Adjacency list

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Code in python:

```
def add_edge(adj, i, j):
    adj[i].append(j)
    adj[j].append(i)  # Undirected

def display_adj_list(adj):
    for i in range(len(adj)):
        print(f"{i}: ", end="")
        for j in adj[i]:
            print(j, end=" ")
        print()

# Create a graph with 4 vertices and no edges
V = 4
adj = [[] for _ in range(V)]

# Now add edges one by one
add_edge(adj, 0, 1)
add_edge(adj, 0, 2)
add_edge(adj, 1, 2)
add_edge(adj, 2, 3)

print("Adjacency List Representation:")
display_adj_list(adj)
```

Code in c++:

```
#include <iostream>
#include <vector>
```

```
using namespace std;

// Function to add an edge between two vertices
void addEdge(vector<vector<int>>& adj, int i, int j) {
    adj[i].push_back(j);
    adj[j].push_back(i); // Undirected
}

// Function to display the adjacency list
void displayAdjList(const vector<vector<int>>& adj) {
    for (int i = 0; i < adj.size(); i++) {
        cout << i << ": "; // Print the vertex
        for (int j : adj[i]) {
            cout << j << " "; // Print its adjacent
        }
        cout << endl;
    }
}

// Main function
int main() {
    // Create a graph with 4 vertices and no edges
    int V = 4;
    vector<vector<int>> adj(V);

    // Now add edges one by one
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 2);
    addEdge(adj, 2, 3);

    cout << "Adjacency List Representation:" << endl;
    displayAdjList(adj);

    return 0;
}
```

Code in javascript:

```
function addEdge(adj, i, j) {
    adj[i].push(j);
    adj[j].push(i); // Undirected
}

function displayAdjList(adj) {
    for (let i = 0; i < adj.length; i++) {
        console.log(`${i}: `);
        for (const j of adj[i]) {
            console.log(`${j} `);
        }
        console.log();
    }
}

// Create a graph with 4 vertices and no edges
const V = 4;
const adj = Array.from({ length: V }, () => []);

// Now add edges one by one
addEdge(adj, 0, 1);
addEdge(adj, 0, 2);
addEdge(adj, 1, 2);
addEdge(adj, 2, 3);

console.log("Adjacency List Representation:");
displayAdjList(adj);
```

Output

```
Adjacency List Representation:
0: 1 2
1: 0 2
2: 0 1 3
3: 2
```

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an edge	O(1)	O(N)

Initializing	O(N*N)	zO(N)

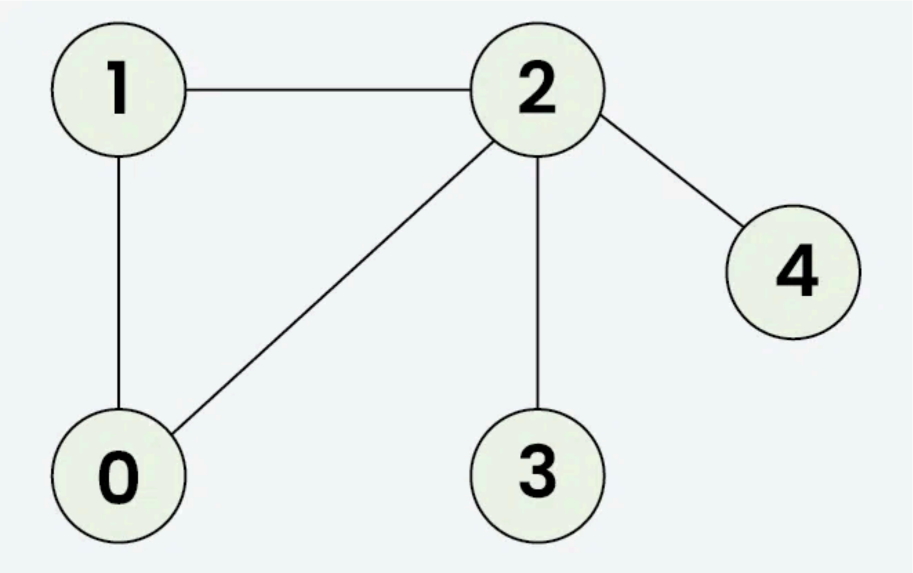
Graph traversal

Depth first search (dfs)

Introduction

In depth first search we traverse all adjacent vertices one by one.
When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex.
There can be multiple DFS traversals of a graph according to the order in which we pick adjacent vertices.
Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.
Auxiliary Space: O(V + E), since an extra visited array of size V is required, And stack size for recursive calls to dfsRec function.

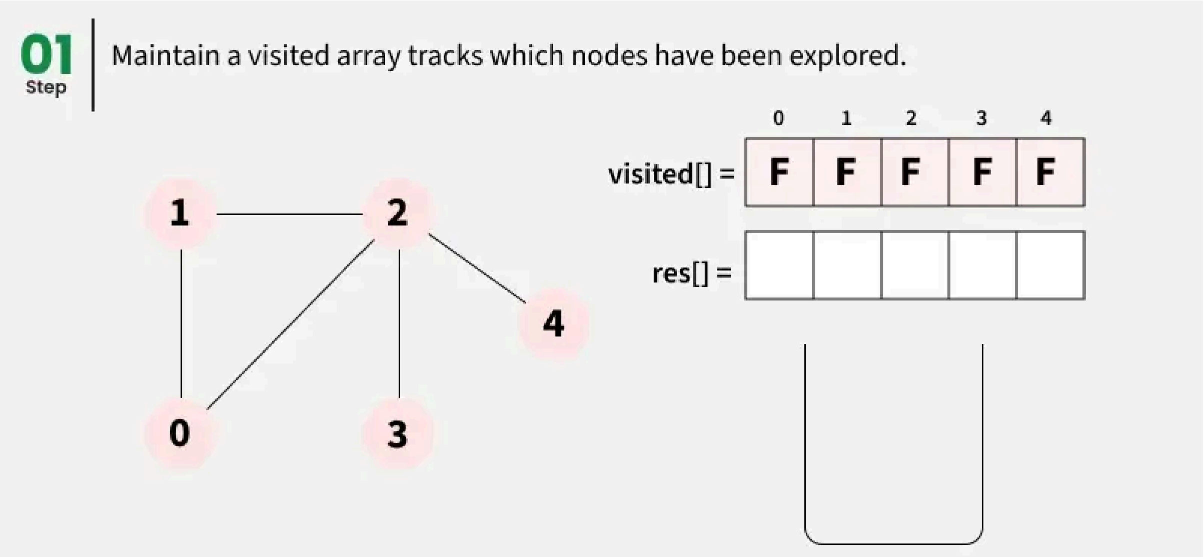
Example,
Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]



Output: [0 1 2 3 4]
Explanation: The source vertex s is 0. We visit it first, then we visit an adjacent.
Start at 0: Mark as visited. Output: 0
Move to 1: Mark as visited. Output: 1

Move to 2: Mark as visited. Output: 2
Move to 3: Mark as visited. Output: 3 (backtrack to 2)
Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 1, then to 0)

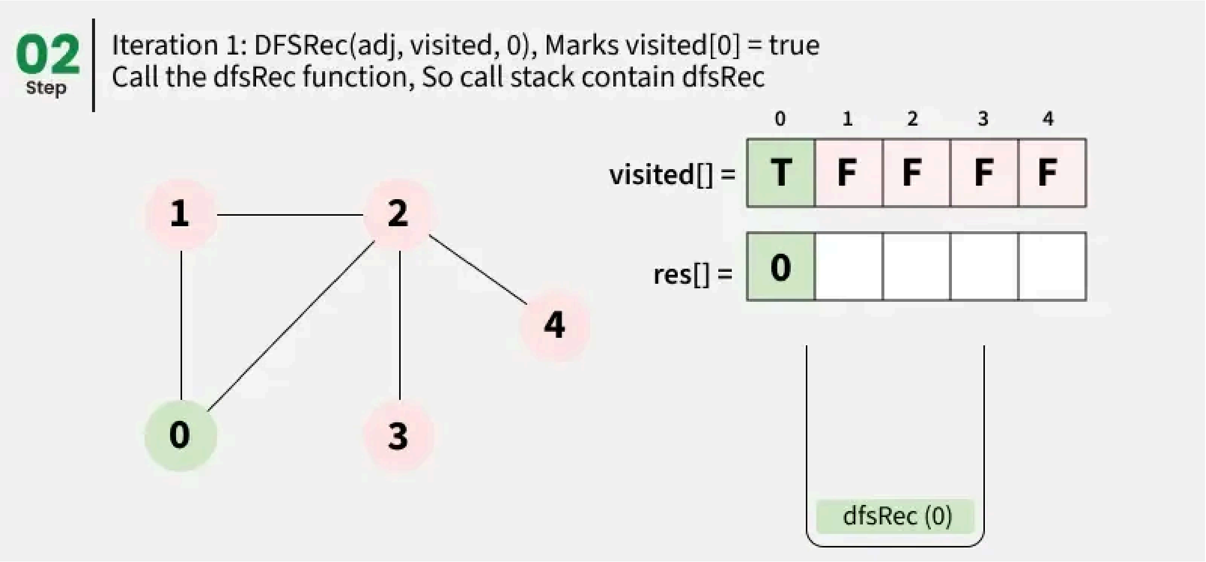
DFS from a Given Source of Undirected Graph



01 Step

Maintain a visited array tracks which nodes have been explored.

	0	1	2	3	4
visited[] =	F	F	F	F	F
res[] =					



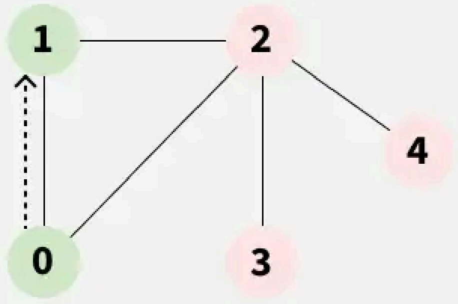
02 Step

Iteration 1: DFSRec(adj, visited, 0), Marks visited[0] = true
Call the dfsRec function, So call stack contain dfsRec

	0	1	2	3	4
visited[] =	T	F	F	F	F
res[] =	0				

03 Step

Iteration 2: DFSRec(adj, visited, 1),
Marks visited[1] = true

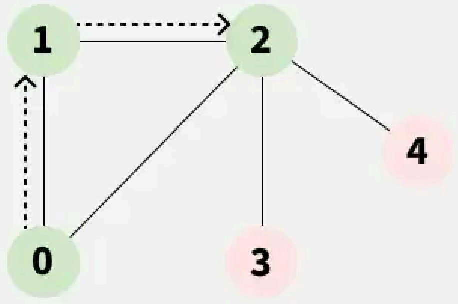


	0	1	2	3	4
visited[] =	T	T	F	F	F
res[] =	0	1			

dfsRec (1)
dfsRec (0)

04 Step

Iteration 3: DFSRec(adj, visited, 2),
Marks visited[2] = true

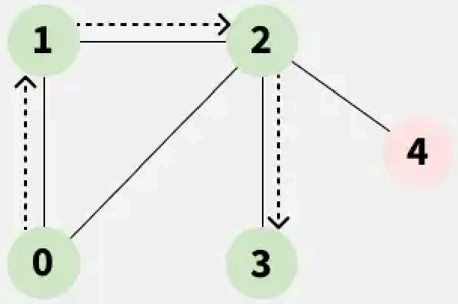


	0	1	2	3	4
visited[] =	T	T	T	F	F
res[] =	0	1	2		

dfsRec (2)
dfsRec (1)
dfsRec (0)

05 Step

Iteration 4: DFSRec(adj, visited,3),
Marks visited[3] = true

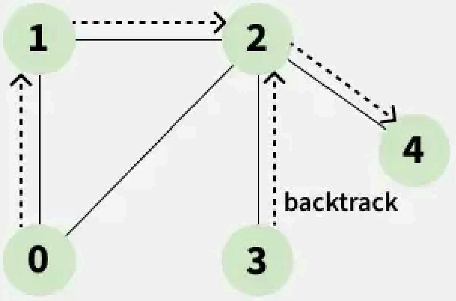


	0	1	2	3	4
visited[] =	T	T	T	T	F
res[] =	0	1	2	3	

dfsRec (3)
dfsRec (2)
dfsRec (1)
dfsRec (0)

06 Step

Iteration 4: DFSRec(adj, visited,3),
Marks visited[3] = true



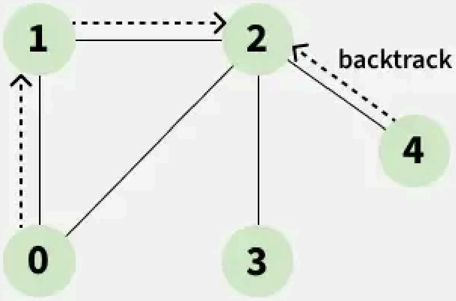
	0	1	2	3	4
visited[] =	T	T	T	T	T

res[] =	0	1	2	3	4

dfsRec (4)
dfsRec (2)
dfsRec (1)
dfsRec (0)

07 Step

Backtracking from 4



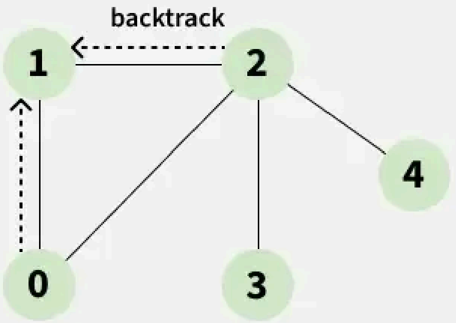
	0	1	2	3	4
visited[] =	T	T	T	T	T

res[] =	0	1	2	3	4

dfsRec (2)
dfsRec (1)
dfsRec (0)

08 Step

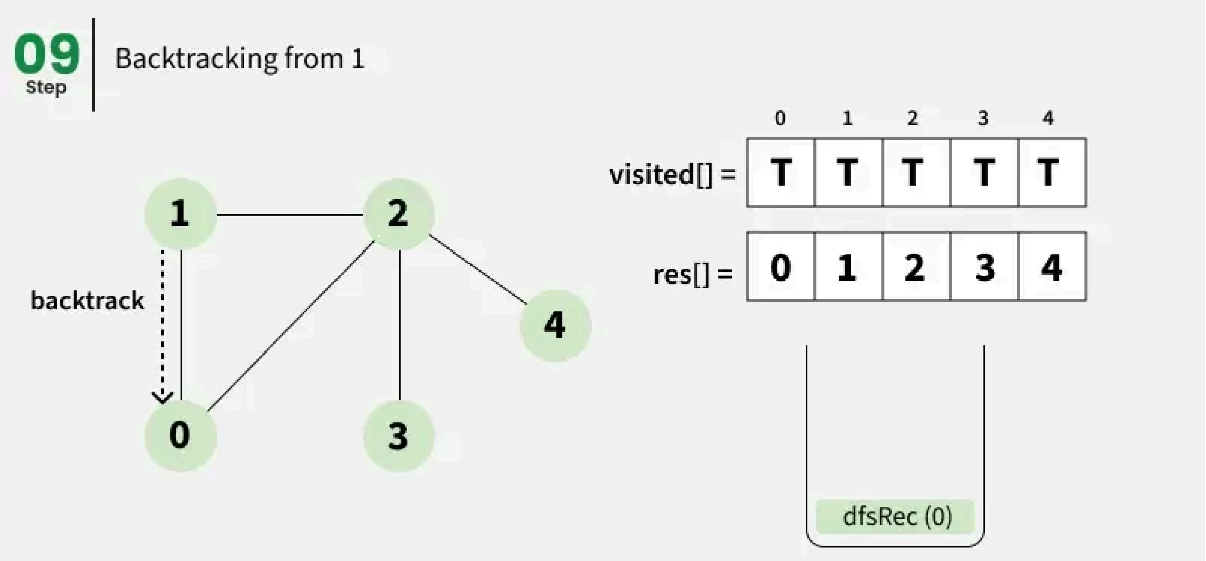
Backtracking from 2



	0	1	2	3	4
visited[] =	T	T	T	T	T

res[] =	0	1	2	3	4

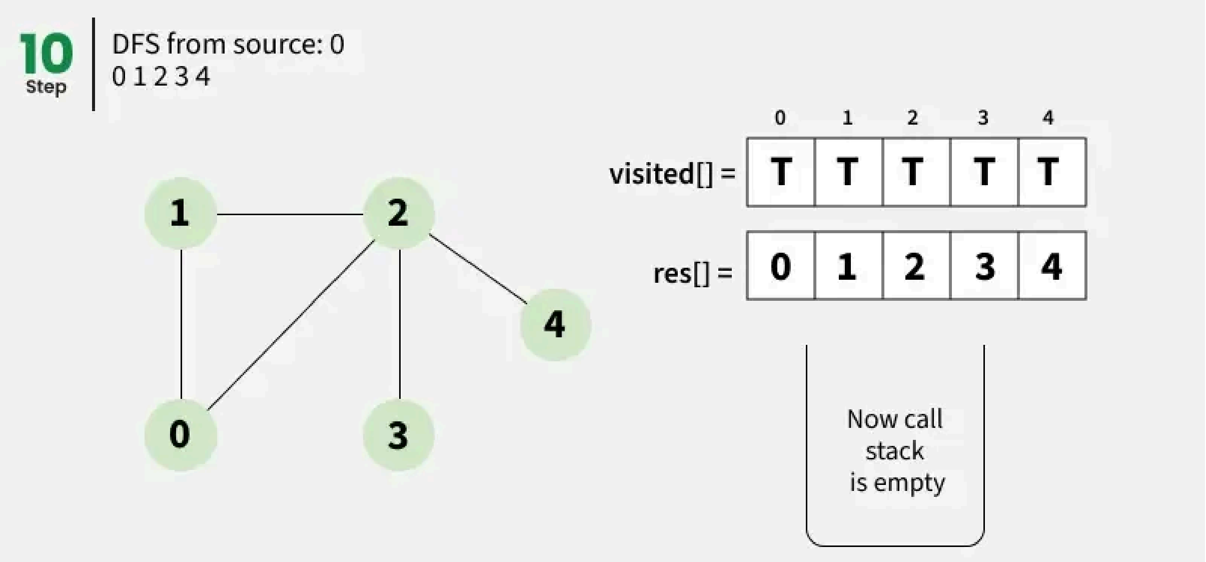
dfsRec (1)
dfsRec (0)



09 Step

Backtracking from 1

	0	1	2	3	4
visited[] =	T	T	T	T	T
res[] =	0	1	2	3	4



10 Step

DFS from source: 0
0 1 2 3 4

	0	1	2	3	4
visited[] =	T	T	T	T	T
res[] =	0	1	2	3	4

Code

Python

```
def dfsRec(adj, visited, s, res):
    visited[s] = True
    res.append(s)

    # Recursively visit all adjacent vertices that are not visited
yet
    for i in range(len(adj)):
        if adj[s][i] == 1 and not visited[i]:
            dfsRec(adj, visited, i, res)


def DFS(adj):
    visited = [False] * len(adj)
    res = []
    dfsRec(adj, visited, 0, res)  # Start DFS from vertex 0
    return res


def add_edge(adj, s, t):
    adj[s][t] = 1
    adj[t][s] = 1  # Since it's an undirected graph


# Driver code
V = 5
adj = [[0] * V for _ in range(V)]  # Adjacency matrix

# Define the edges of the graph
edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]

# Populate the adjacency matrix with edges
for s, t in edges:
    add_edge(adj, s, t)

res = DFS(adj)  # Perform DFS
print(" ".join(map(str, res)))
```

C++

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

// Recursive function for DFS traversal
void dfsRec(vector<vector<int>> &adj, vector<bool> &visited, int
s, vector<int> &res)
{

    visited[s] = true;

    res.push_back(s);

    // Recursively visit all adjacent vertices
    // that are not visited yet
    for (int i : adj[s])
        if (visited[i] == false)
            dfsRec(adj, visited, i, res);
}

// Main DFS function that initializes the visited array
// and call DFSRec
```

```
vector<int> DFS(vector<vector<int>> &adj)
{
    vector<bool> visited(adj.size(), false);
    vector<int> res;
    dfsRec(adj, visited, 0, res);
    return res;
}

// To add an edge in an undirected graph
void addEdge(vector<vector<int>> &adj, int s, int t)
{
    adj[s].push_back(t);
    adj[t].push_back(s);
}

int main()
{
    int V = 5;
    vector<vector<int>> adj(V);

    // Add edges
    vector<vector<int>> edges = {{1, 2}, {1, 0}, {2, 0}, {2, 3},
{2, 4}};
    for (auto &e : edges)
        addEdge(adj, e[0], e[1]);

    // Starting vertex for DFS
    vector<int> res = DFS(adj); // Perform DFS starting from the
source verte 0;

    for (int i = 0; i < V; i++)
        cout << res[i] << " ";
}
```

Javascript

```
function dfsRec(adj, visited, s, res)
{
    visited[s] = true;
    res.push(s);

    // Recursively visit all adjacent vertices that are not
    // visited yet
    for (let i = 0; i < adj.length; i++) {
        if (adj[s][i] === 1 && !visited[i]) {
            dfsRec(adj, visited, i, res);
        }
    }
}

function DFS(adj)
{
    let visited = new Array(adj.length).fill(false);
    let res = [];
    dfsRec(adj, visited, 0, res); // Start DFS from vertex 0
    return res;
}

function addEdge(adj, s, t)
{
    adj[s][t] = 1;
    adj[t][s] = 1; // Since it's an undirected graph
}

// Driver code
let V = 5;
let adj = Array.from(
    {length : V},
    () => new Array(V).fill(0)); // Adjacency matrix

// Define the edges of the graph
let edges =
    [ [ 1, 2 ], [ 1, 0 ], [ 2, 0 ], [ 2, 3 ], [ 2, 4 ] ];

// Populate the adjacency matrix with edges
edges.forEach(([ s, t ]) => addEdge(adj, s, t));

let res = DFS(adj); // Perform DFS
console.log(res.join(" "));
```

Output

```
0 1 2 3 4
```

DFS for Complete Traversal of Disconnected Undirected Graph

The idea is simple, instead of calling DFS for a single vertex, we call the above implemented DFS for all non-visited vertices one by one.
Time complexity: O(V + E). Note that the time complexity is same here because we visit every vertex at most once and every edge is traversed at most once (in directed) and twice in undirected.
Auxiliary Space: O(V + E), since an extra visited array of size V is required, And stack size for recursive calls to dfsRec function.

Code

Python

```
# Create an adjacency list for the graph
from collections import defaultdict


def add_edge(adj, s, t):
    adj[s].append(t)
    adj[t].append(s)

# Recursive function for DFS traversal


def dfs_rec(adj, visited, s, res):
    # Mark the current vertex as visited
    visited[s] = True
    res.append(s)

    # Recursively visit all adjacent vertices that are not visited
yet
    for i in adj[s]:
        if not visited[i]:
            dfs_rec(adj, visited, i, res)

# Main DFS function to perform DFS for the entire graph


def dfs(adj):
    visited = [False] * len(adj)
    res = []
    # Loop through all vertices to handle disconnected graph
    for i in range(len(adj)):
        if not visited[i]:
            # If vertex i has not been visited,
            # perform DFS from it
            dfs_rec(adj, visited, i, res)
    return res


V = 6
# Create an adjacency list for the graph
adj = defaultdict(list)

# Define the edges of the graph
edges = [[1, 2], [2, 0], [0, 3], [4, 5]]

# Populate the adjacency list with edges
```

```
for e in edges:
    add_edge(adj, e[0], e[1])
res = dfs(adj)

print(' '.join(map(str, res)))
```

C++

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

void addEdge(vector<vector<int>> &adj, int s, int t)
{
    adj[s].push_back(t);
    adj[t].push_back(s);
}

// Recursive function for DFS traversal
void dfsRec(vector<vector<int>> &adj, vector<bool> &visited, int
s, vector<int> &res)
{
    // Mark the current vertex as visited
    visited[s] = true;

    res.push_back(s);

    // Recursively visit all adjacent vertices that are not
visited yet
    for (int i : adj[s])
        if (visited[i] == false)
            dfsRec(adj, visited, i, res);
}

// Main DFS function to perform DFS for the entire graph
vector<int> DFS(vector<vector<int>> &adj)
{
    vector<bool> visited(adj.size(), false);
    vector<int> res;
    // Loop through all vertices to handle disconnected graph
    for (int i = 0; i < adj.size(); i++)
    {
        if (visited[i] == false)
        {
            // If vertex i has not been visited,
            // perform DFS from it
            dfsRec(adj, visited, i, res);
        }
    }
    return res;
}

int main()
{
    int V = 6;
    // Create an adjacency list for the graph
    vector<vector<int>> adj(V);

    // Define the edges of the graph
    vector<vector<int>> edges = {{1, 2}, {2, 0}, {0, 3}, {4, 5}};

    // Populate the adjacency list with edges
    for (auto &e : edges)
        addEdge(adj, e[0], e[1]);
    vector<int> res = DFS(adj);

    for (auto it : res)
        cout << it << " ";
    return 0;
}
```

Javascript

```
function addEdge(adj, s, t) {
    adj[s].push(t);
    adj[t].push(s);
}

// Recursive function for DFS traversal
function dfsRec(adj, visited, s, res) {
    visited[s] = true;
    res.push(s);

    // Recursively visit all adjacent vertices that are not
visited yet
    for (let i of adj[s]) {
        if (!visited[i]) {
            dfsRec(adj, visited, i, res);
        }
    }
}

// Main DFS function to perform DFS for the entire graph
function DFS(adj) {
    let visited = new Array(adj.length).fill(false);
    let res = [];

    // Loop through all vertices to handle disconnected graphs
    for (let i = 0; i < adj.length; i++) {
        if (!visited[i]) {
            dfsRec(adj, visited, i, res);
        }
    }

    return res;
}

// Main Execution
let V = 6;
// Create an adjacency list for the graph
let adj = Array.from({ length: V }, () => []);

let edges = [[1, 2], [2, 0], [0, 3], [4, 5]];

// Populate the adjacency list with edges
for (let e of edges) {
    addEdge(adj, e[0], e[1]);
}

// Perform DFS
let res = DFS(adj);

// Print the DFS traversal result
console.log(res.join(" "));
```

Output

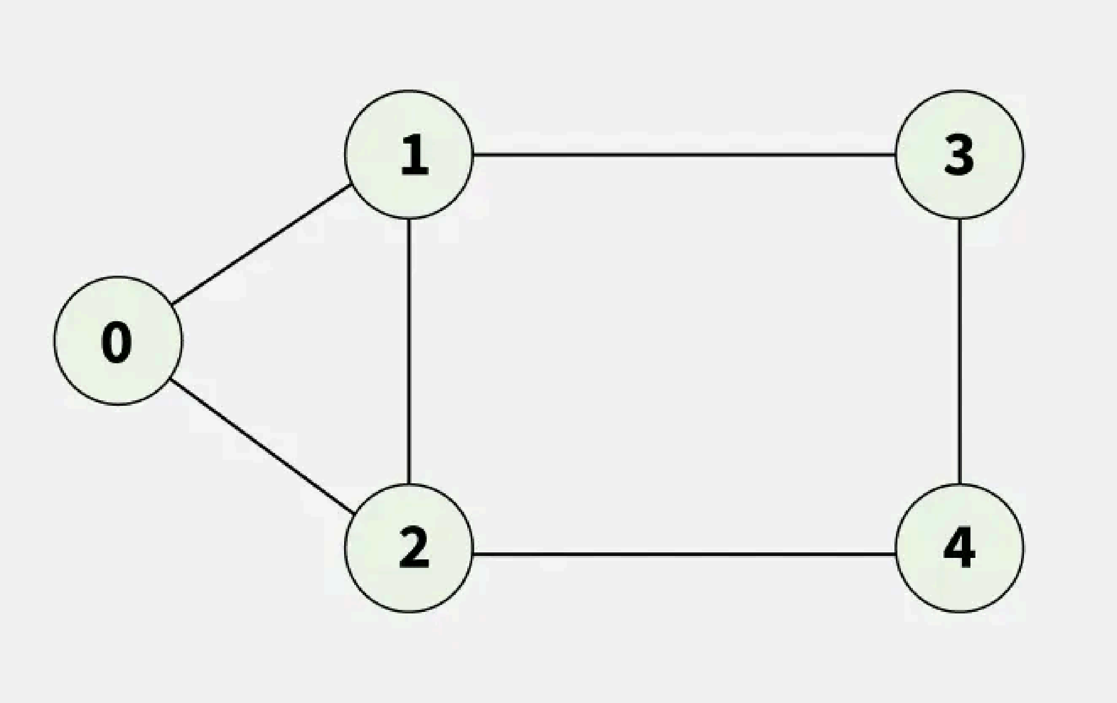
```
0 2 1 3 4 5
```

Breadth first search (bfs)

Introduction

Breadth First Search visits all adjacent vertices of a vertex before visiting neighboring vertices to the adjacent vertices.
This means that vertices with the same distance from the starting vertex are visited before vertices further away from the starting vertex are visited.
Time Complexity: O(V + E)
Auxiliary Space: O(V)

Example,
Input: adj[][] = [[1,2], [0,2,3], [0,1,4], [1,4], [2,3]]



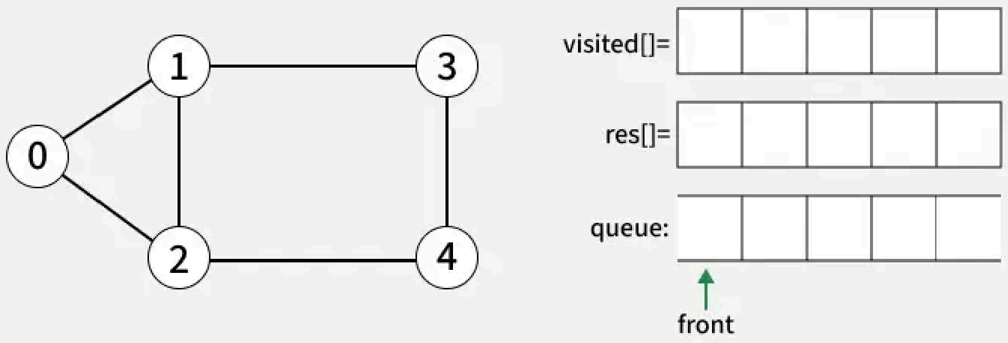
Output: [0, 1, 2, 3, 4]
Explanation: Starting from 0, the BFS traversal will follow these steps:
Visit 0 → Output: [0]
Visit 1 (first neighbor of 0) → Output: [0, 1]
Visit 2 (next neighbor of 0) → Output: [0, 1, 2]
Visit 3 (next neighbor of 1) → Output: [0, 1, 2, 3]
Visit 4 (neighbor of 2) → Final Output: [0, 1, 2, 3, 4]

BFS from a Given Source

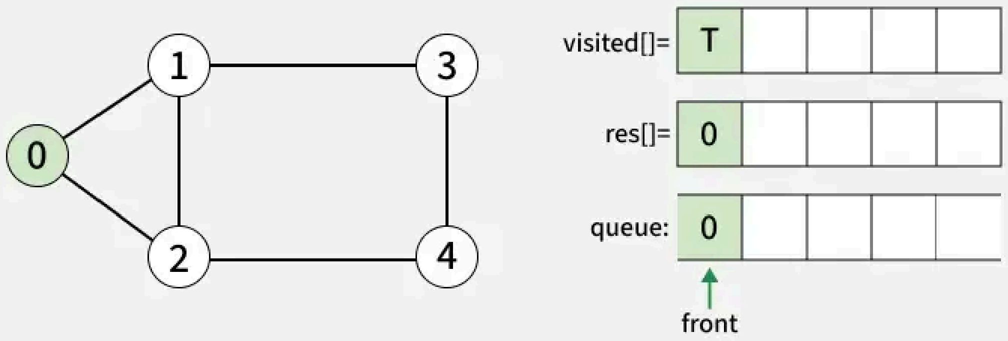
Approach:

1. **Initialization:** Enqueue the given source vertex into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
    - Enqueue the neighbor into the queue.
    - Mark the neighbor as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

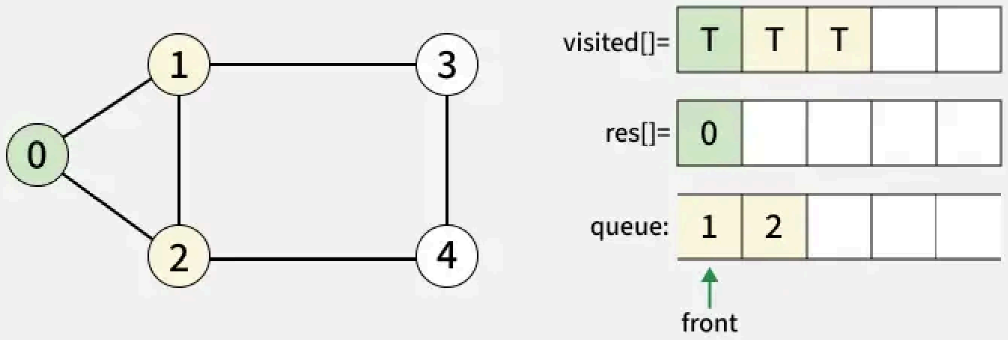
01 Maintain a queue, res[], and a Visited array, such that the queue stores nodes to be processed in FIFO order, res[] stores the BFS traversal sequence, and Visited keeps track of visited nodes to prevent reprocessing



02 Start the traversal with node 0, push it into the queue, mark it as visited, and while the queue is not empty, pop the front node, store it in res

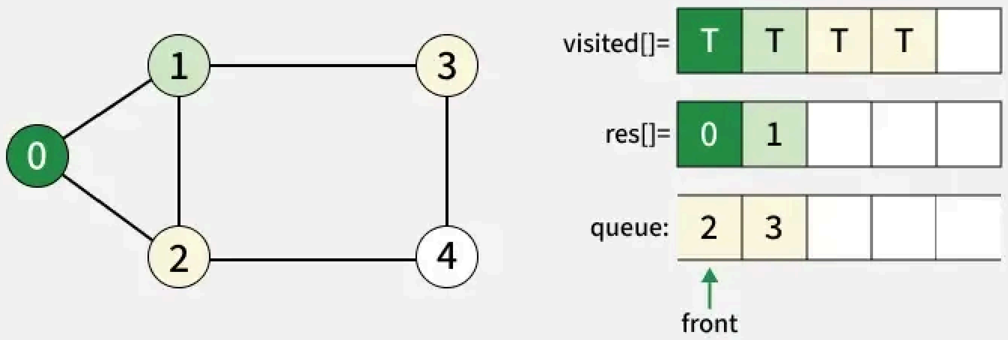


03 Remove the front element 0 from the queue, store it in res, visit its unvisited neighbors, mark them as visited, and push them into the queue for further processing.



04

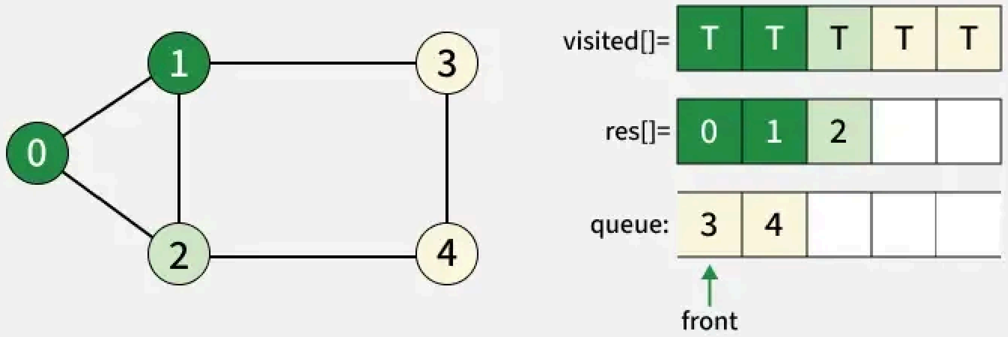
Remove the front element 1 from the queue, store it in res, visit its unvisited neighbors, mark them as visited, and push them into the queue for further processing.



visited[]=	T	T	T	T	
res[]=	0	1			
queue:	2	3			

05

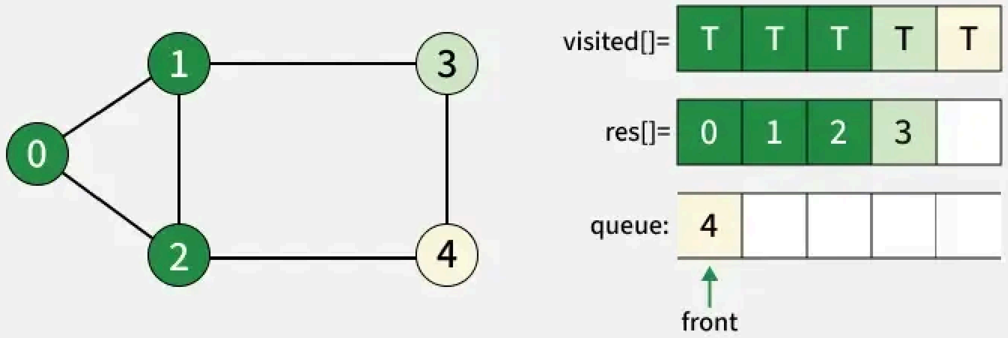
Remove the front element 2 from the queue, store it in res, visit its unvisited neighbors, mark them as visited, and push them into the queue for further processing.



visited[]=	T	T	T	T	T
res[]=	0	1	2		
queue:	3	4			

06

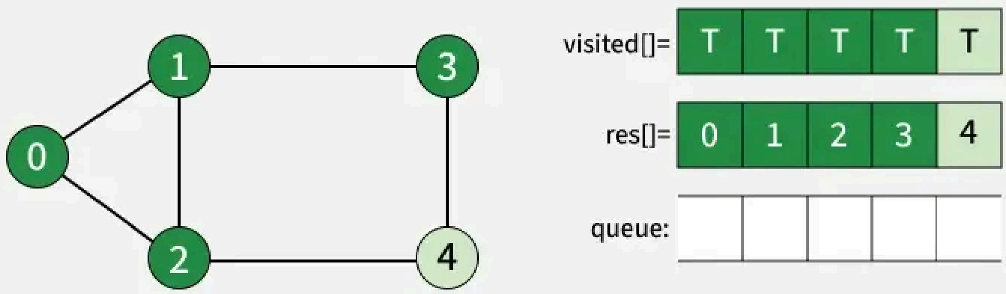
Remove the front element 3 from the queue, store it in res, visit its unvisited neighbors, mark them as visited, and push them into the queue for further processing.



visited[]=	T	T	T	T	T
res[]=	0	1	2	3	
queue:	4				

All the node of 3 have been visited, proceed to the next node in the queue

07 Remove the front element 4 from the queue, store it in res, visit its unvisited neighbors, mark them as visited, and push them into the queue for further processing.



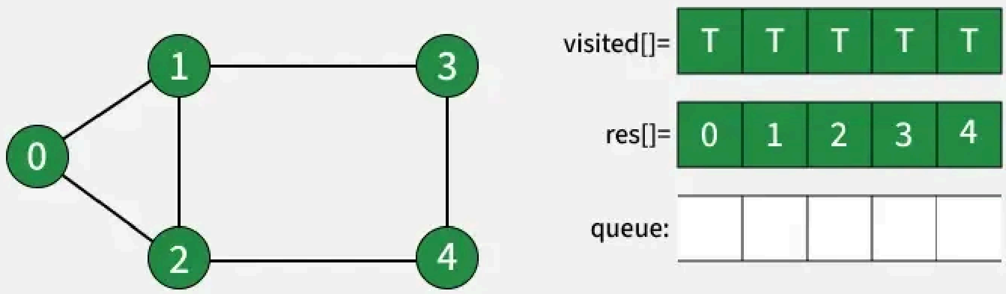
visited[]= T T T T T

res[]= 0 1 2 3 4

queue:

All the node of 4 have been visited, proceed to the next node in the queue

08 Now that the queue is empty and there are no more nodes to process, we obtain the final BFS traversal.



visited[]= T T T T T

res[]= 0 1 2 3 4

queue:

final BFS traversal order is 0, 1, 2, 3, 4.

Code

Python

```
# Function to find BFS of Graph from given source s
def bfs(adj):

    # get number of vertices
    V = len(adj)

    # create an array to store the traversal
    res = []
    s = 0
    # Create a queue for BFS
    from collections import deque
    q = deque()

    # Initially mark all the vertices as not visited
    visited = [False] * V
    # Mark source node as visited and enqueue it
    visited[s] = True
    q.append(s)

    # Iterate over the queue
    while q:

        # Dequeue a vertex from queue and store it
        curr = q.popleft()
        res.append(curr)

        # Get all adjacent vertices of the dequeued
        # vertex curr If an adjacent has not been
        # visited, mark it visited and enqueue it
        for x in adj[curr]:
            if not visited[x]:
                visited[x] = True
                q.append(x)

    return res

if __name__ == "__main__":

    # create the adjacency list
    # [ [2, 3, 1], [0], [0, 4], [0], [2] ]
    adj = [[1,2], [0,2,3], [0,4], [1,4], [2,3]]
    ans = bfs(adj)
    for i in ans:
        print(i, end=" ")
```

C++

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

// BFS from given source s
vector<int> bfs(vector<vector<int>>& adj)  {
    int V = adj.size();

    int s = 0; // source node
    // create an array to store the traversal
    vector<int> res;

    // Create a queue for BFS
    queue<int> q;

    // Initially mark all the vertices as not visited
    vector<bool> visited(V, false);
    // Mark source node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (!q.empty()) {

        // Dequeue a vertex from queue and store it
        int curr = q.front();
        q.pop();
        res.push_back(curr);

        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }

    return res;
}

int main()  {

    vector<vector<int>> adj = {{1,2}, {0,2,3}, {0,4}, {1,4},
{2,3}};
    vector<int> ans = bfs(adj);
    for(auto i:ans) {
        cout<<i<<" ";
    }
    return 0;
}
```

Javascript

```
// Function to find BFS of Graph from given source s
function bfs(adj) {
    let V = adj.length;
    let s = 0; // source node is 0
    // create an array to store the traversal
    let res = [];

    // Create a queue for BFS
    let q = [];
    // Initially mark all the vertices as not visited
    let visited = new Array(V).fill(false);

    // Mark source node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (q.length > 0) {

        // Dequeue a vertex from queue and store it
        let curr = q.shift();
        res.push(curr);

        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (let x of adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
    return res;
}

// Main execution
let adj =
    [ [1,2], [0,2,3], [0,4], [1,4], [2,3]];
let src = 0;
let ans = bfs(adj);
for (let i of ans) {
    process.stdout.write(i + " ");
}
```

Output

```
0 1 2 3 4
```

BFS of the Disconnected Graph

The above implementation takes a source as an input and prints only those vertices that are reachable from the source and would not print all vertices in case of disconnected graph.

Code

Python

```
from collections import deque

def bfsOfGraph(adj, s, visited, res):

    # Create a queue for BFS
    q = deque()

    # Mark source node as visited and enqueue it
    visited[s] = True
    q.append(s)

    # Iterate over the queue
    while q:

        # Dequeue a vertex and store it
        curr = q.popleft()
        res.append(curr)

        # Get all adjacent vertices of the dequeued
        # vertex curr If an adjacent has not been
        # visited, mark it visited and enqueue it
        for x in adj[curr]:
            if not visited[x]:
                visited[x] = True
                q.append(x)
    return res

# Perform BFS for the entire graph which maybe
# disconnected
def bfsDisconnected(adj):
    V = len(adj)

    # create an array to store the traversal
    res = []

    # Initially mark all the vertices as not visited
    visited = [False] * V

    # perform BFS for each node
    for i in range(V):
        if not visited[i]:
            bfsOfGraph(adj, i, visited, res)
    return res

if __name__ == "__main__":
    adj = [[1, 2], [0], [0],
        [4], [3, 5], [4]]
    ans = bfsDisconnected(adj)
    for i in ans:
        print(i, end=" ")
```

C++

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

// BFS from given source s
void bfs(vector<vector<int>>& adj, int s,
        vector<bool>& visited, vector<int> &res) {

    // Create a queue for BFS
    queue<int> q;

    // Mark source node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (!q.empty()) {

        // Dequeue a vertex and store it
        int curr = q.front();
        q.pop();
        res.push_back(curr);

        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
}

// Perform BFS for the entire graph which maybe
// disconnected
vector<int> bfsDisconnected(vector<vector<int>>& adj) {
    int V = adj.size();

    // create an array to store the traversal
    vector<int> res;
    // Initially mark all the vertices as not visited
    vector<bool> visited(V, false);

    // perform BFS for each node
    for (int i = 0; i < adj.size(); ++i) {
        if (!visited[i]) {
            bfs(adj, i, visited, res);
        }
    }

    return res;
}

int main()  {

    vector<vector<int>> adj = { {1, 2}, {0}, {0},
                                {4}, {3, 5}, {4}};
    vector<int> ans = bfsDisconnected(adj);
    for(auto i:ans) {
        cout<<i<<" ";
    }
    return 0;
}
```

Javascript

```
function bfsOfGraph(adj, s, visited, res) {

    // Create a queue for BFS
    let q = [];

    // Mark source node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (q.length > 0) {

        // Dequeue a vertex and store it
        let curr = q.shift();
        res.push(curr);

        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (let x of adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
    return res;
}

// Perform BFS for the entire graph which maybe
// disconnected
function bfsDisconnected(adj) {
    let V = adj.length;

    // create an array to store the traversal
    let res = [];

    // Initially mark all the vertices as not visited
    let visited = new Array(V).fill(false);

    // perform BFS for each node
    for (let i = 0; i < V; i++) {
        if (!visited[i]) {
            bfsOfGraph(adj, i, visited, res);
        }
    }
    return res;
}

// Main execution
let adj =
    [[1, 2], [0], [0],
    [4], [3, 5], [4]];
let ans = bfsDisconnected(adj);
for (let i of ans) {
    process.stdout.write(i + " ");
}
```

Output

```
0 1 2 3 4 5
```