GRAPHS

Two Type of Traversals

1. **BFS** → Breadth First Search
2. **DFS** → Depth First Search

Breadth First Search (BFS)

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS Algorithm

A standard BFS implementation puts each vertex of the graph into one of the two categories:

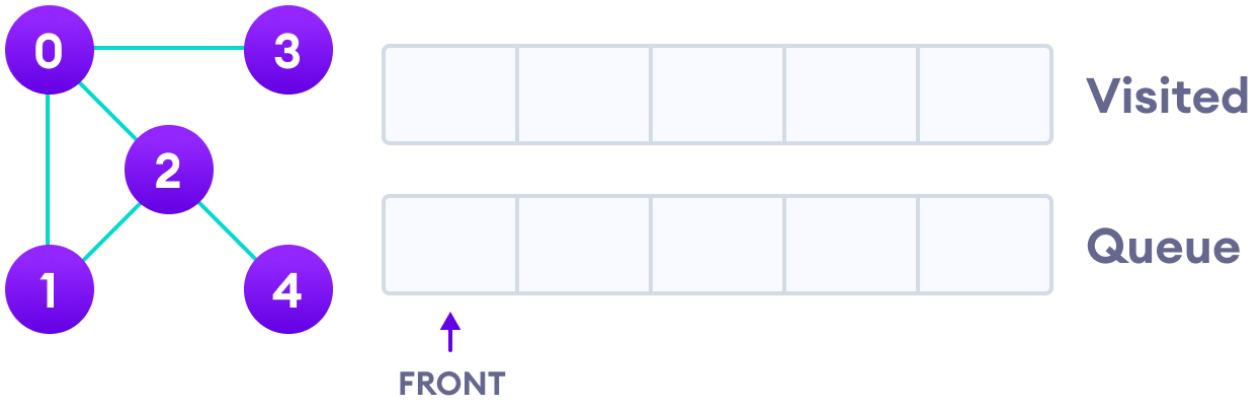
1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. The algorithms work as follows:

a) Start by putting any one of the graph's vertices at the back of a queue
b) Take the front item of the queue and add it to the visited list.
c) Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
d) Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node.

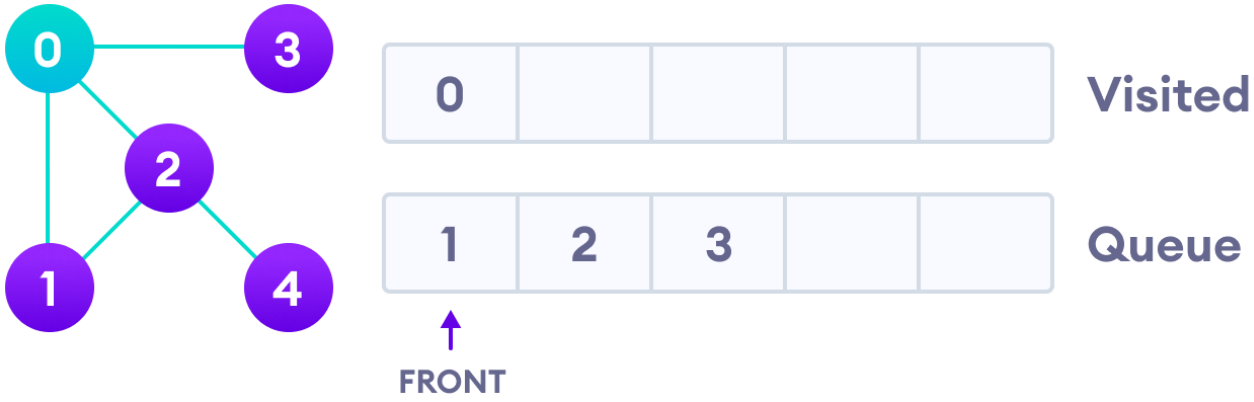
BFS



Example

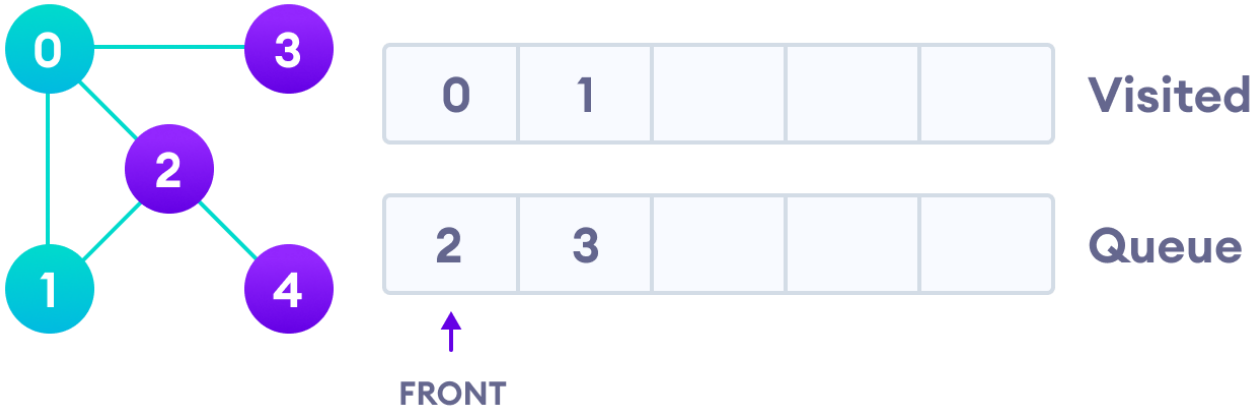
Undirected graph with 5 vertices

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



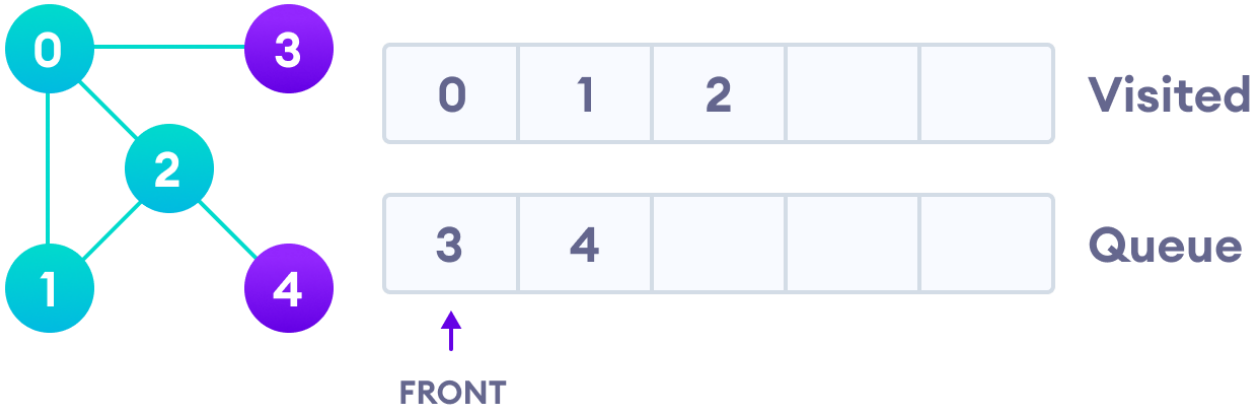
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

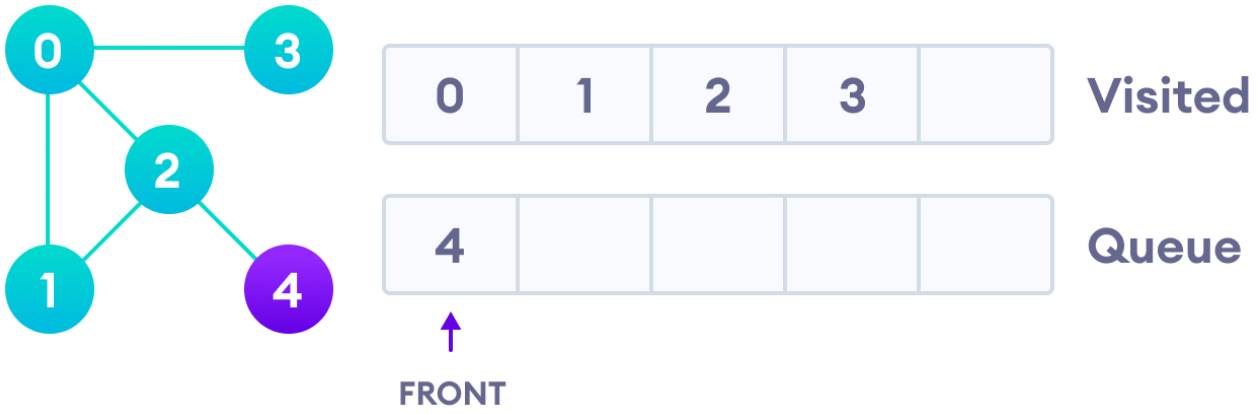


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



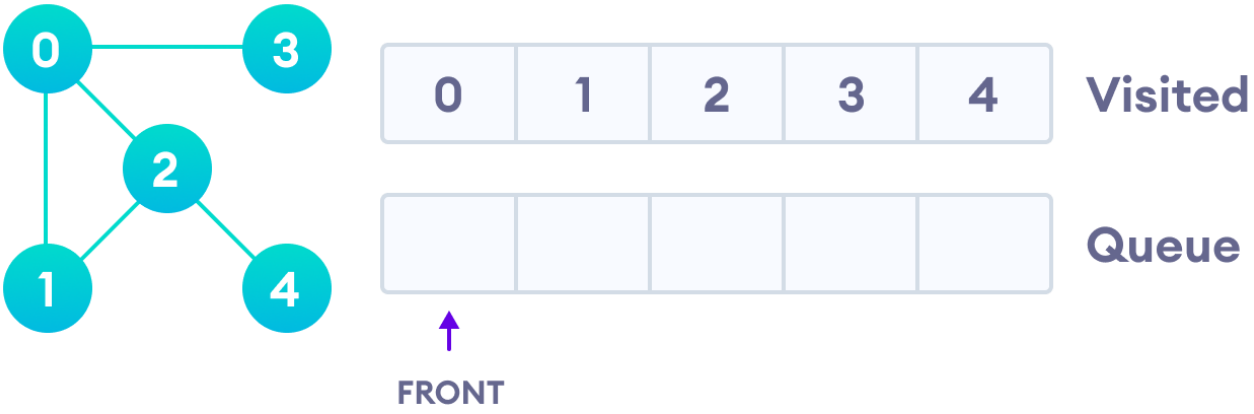
Visit 2 which was added to queue earlier to add its



neighbours

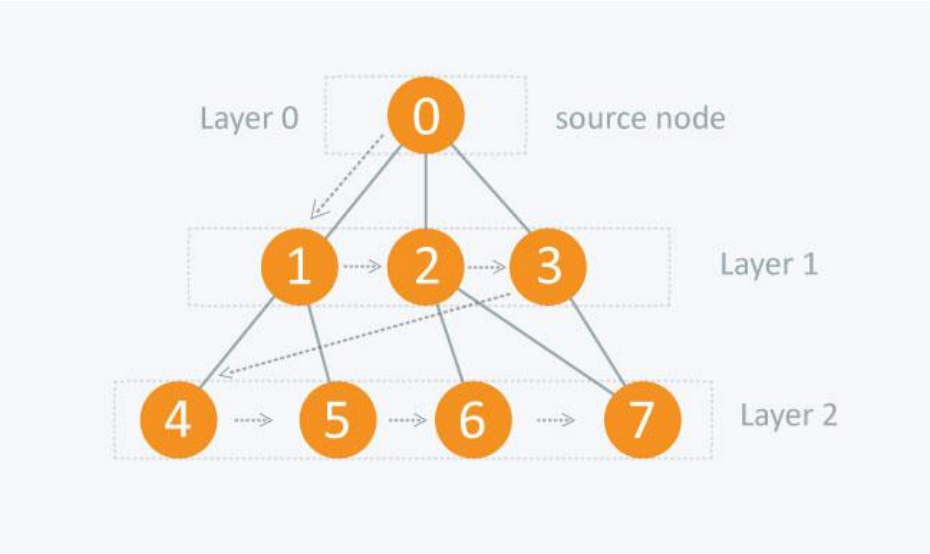
4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e., 0 is already visited. We visit it.



Visit last remaining item in the stack to check if it has unvisited neighbours

Since the queue is empty, we have completed the Breadth First Traversal of the graph.



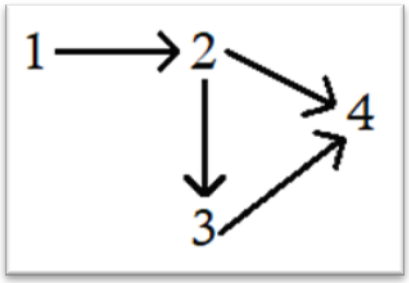
BFS gets printed in Level Order Traversal manner by printing the first level then second level and the last level.

In the dia, layer 0 having 0 gets printed first and then layer 1 having 1 2 & 3 will get printed. Finally, layer 2 having 4 5 6 & 7 will get printed.

[0, 1, 2, 3, 4, 5, 6, 7]

Problems:

Q)



Queue = [ ]
Visited = [F F F F F]
0 1 2 3 4

This will have index from 0 to 4. The visited array will keep track of all the index that are visited. So, for the first element 1, the value at 1st index would become True and it will be enqueued to the queue array. So,

Queue = [1] & Visited = [F T F F F]

For 2, the value at the 2nd index would become True and 2 will be enqueued to the queue array. So,

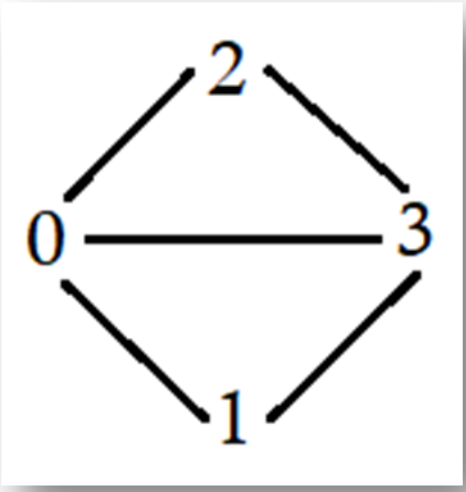
Queue = [1, 2] & Visited = [F T T F F]

For 3, the value at the 3rd index would become True and 3 will be enqueued to the queue array. So,

Queue = [1, 2, 3] & Visited = [F T T T F]

For 4, the value at the 4th index would become True and 4 will be enqueued to the queue array. So,

Queue = [1, 2, 3, 4] & Visited = [F T T T T]



Q) This is an un-directed graph, so we will create a queue and another array, "visited". Start from '2',

Queue = [ ] & Visited = [F F F F]

Queue = [2] & Visited = [F F T F]

Queue = [2, 0] & Visited = [T F T F]

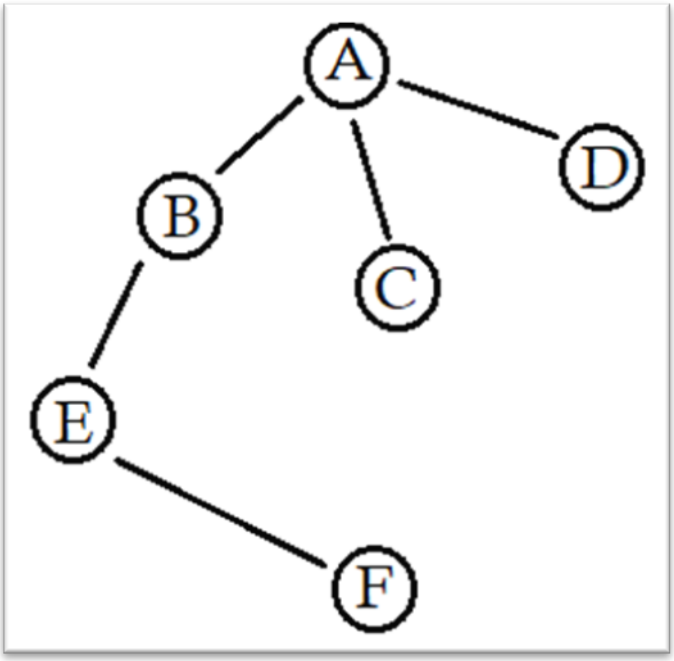
Queue = [2, 0, 3] & Visited = [T F T T]

Queue = **[2, 0, 3, 1]** & Visited = [T T T T]

CODE:

```
graph = dict()

def bfs(src):
    visited = [False] * 10005
    queue = []

    queue.append(src)
    visited[src] = True

    while len(queue) != 0:
        x = queue.pop(0)
        print(x)

        for neibour, wt in graph[x]:
            if not visited[neibour]:
                queue.append(neibour)
                visited[neibour] = True

def addEdge(u, v, weight, directed):
    if u not in graph:
        graph[u] = list()

    graph[u].append((v, weight))

    if not directed:
        if v not in graph:
            graph[v] = list()
        graph[v].append((u, weight))

if __name__ == "__main__":
    addEdge(1, 2, 1, False)
    addEdge(0, 3, 10, False)
    addEdge(2, 3, 8, False)
    addEdge(3, 2, 11, False)
    addEdge(2, 5, 133, False)


    bfs(0)
```

For V vertices and E edges in a graph, the time complexity would be ***O(V + E)***



In this graph we have 6 vertices and 5 edges.

V = 6 E = 5

In this algorithm, we are touching each vertex once and then we are putting neighbors in the queue.

A has 3 edges
B has 1 edge
C & D has 0 edges
E has 1 edge
F has 0 edge

So, for each of these vertices we have V work and for edges, as well we have E works hence, Time Complexity ➔ ***O(V + E)***.

Depth First Search (DFS)

Depth First Search or Depth First Traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

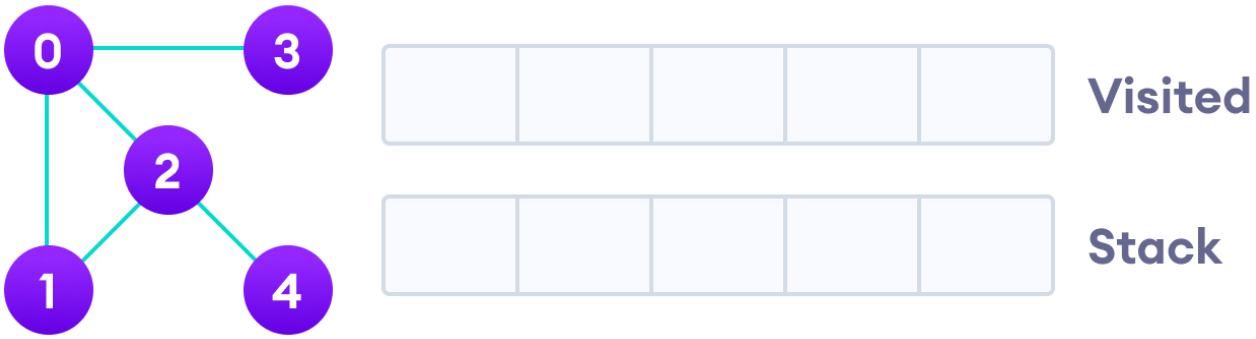
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. The algorithms work as follows:

The DFS algoritm works as follows:

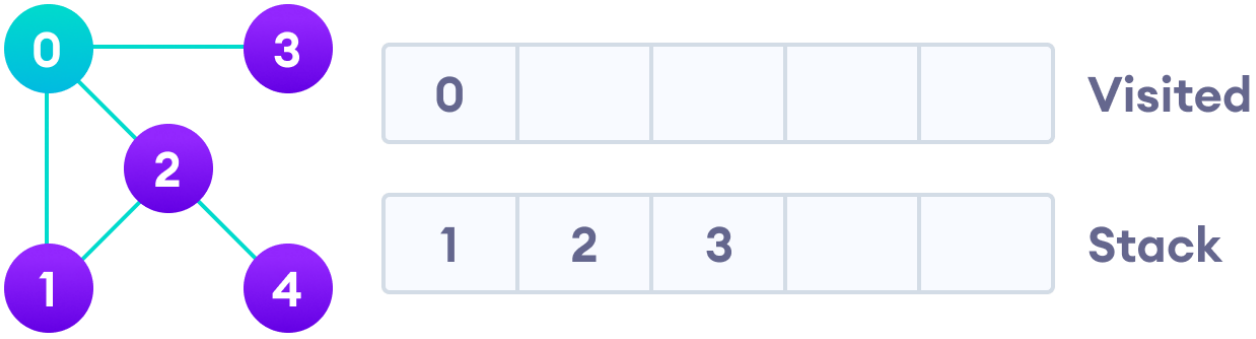
a) Start by putting any one of the graph's vertices on top of a stack
b) Take the top item of the stack and add it to the visited list.
c) Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
d) Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search Example

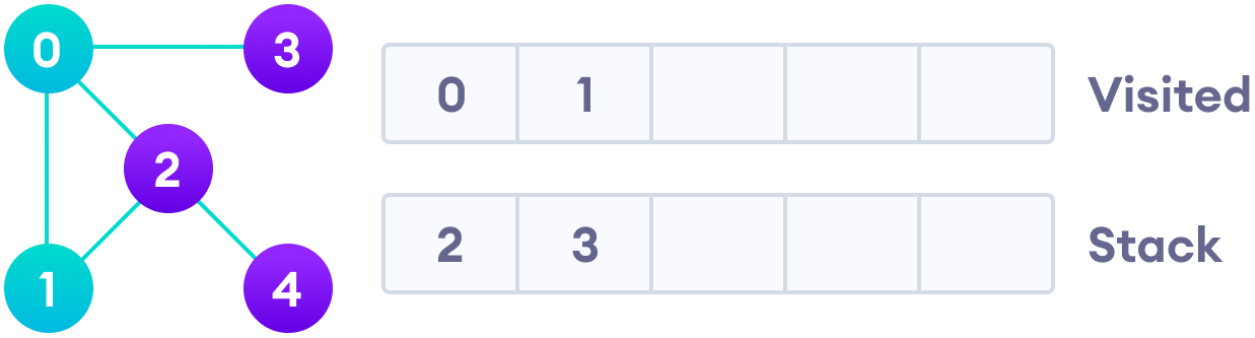
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



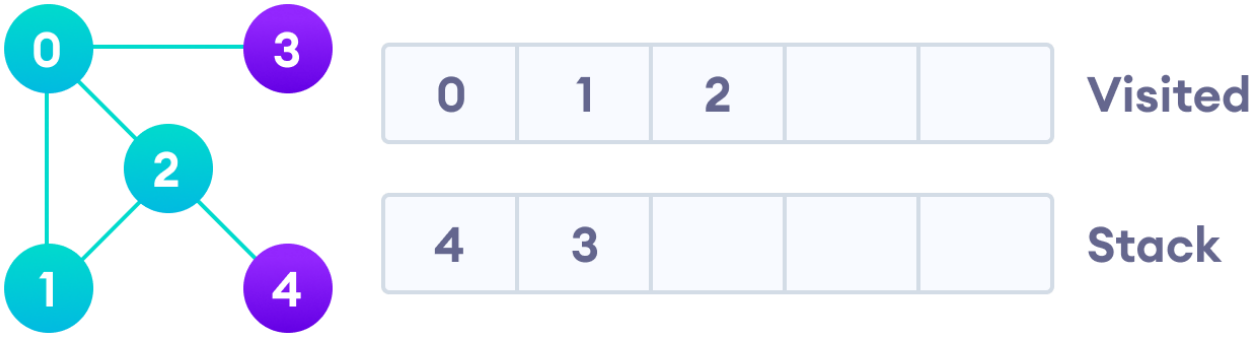
Undirected graph with 5 vertices. We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



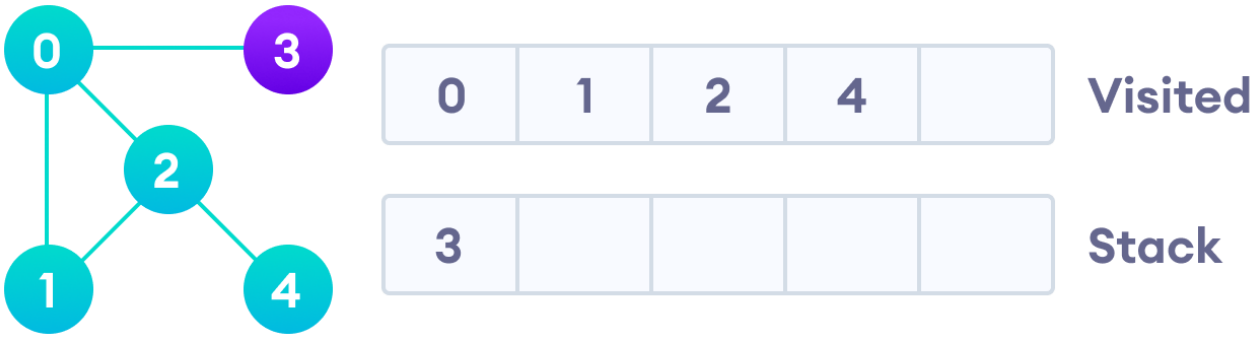
Visit the element and put it in the visited list. Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has been visited, we visit 2 instead.



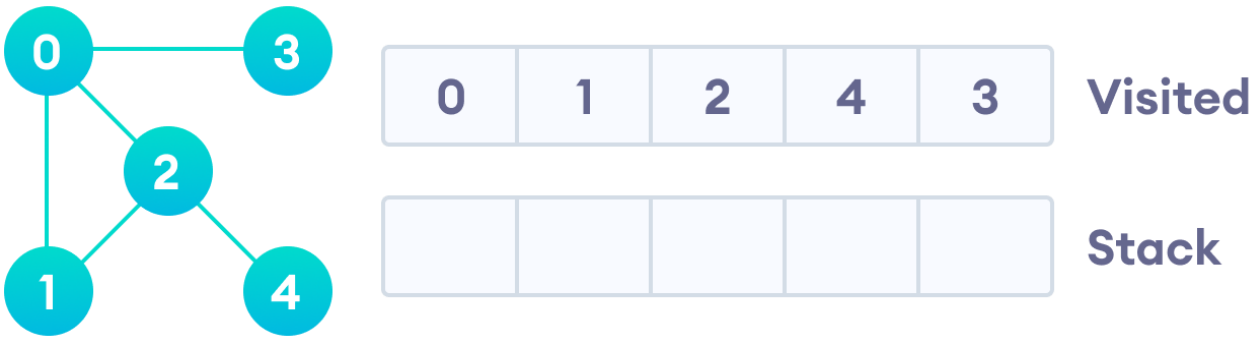
Visit the element at the top of stack. Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

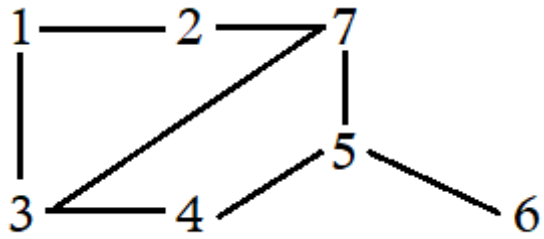


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it. After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

Ex: It is based on stacks and recursion. The size of the visited list would be 7



Queue = [ ]

Visited = [F F F F F F F F]
0 1 2 3 4 5 6 7

We will start our traverse from 1. Since we have visited 1, in our visited array the element at 1st index becomes True.

Visited = [F **T** F F F F F F]

2 & 3 are neighbors of 1. Now, we will first explore the path from 2 so we will visit 2 first and the 2nd index element is updated to True.

Visited = [F **T** **T** F F F F F]

Now, the neighbors of 2 are 1 & 7. Since 1 is already visited, we will visit 7 and the value at the 7th index in visited array is updated to True.

Visited = [F **T** **T** F F F F **T**]

Now, the neighbors of 7 are 3, 5 & 6. 3 has not been visited and we will visit 3 and the value at the 3rd index in the visited array is updated to True.

Visited = [F **T** **T** **T** F F F **T**]

The neighbors of 3 are, 4 & 7. Among them, 7 has been visited and hence we will visit 4 and the value at the 4th index in the visited array is updated to True.

Visited = [F **T** **T** **T** **T** F F **T**]

The neighbors of 4 are. 5 & 3. Among them, 3 has been visited and hence we will visit 5 and the value at the 5th index in the visited array is updated to True.

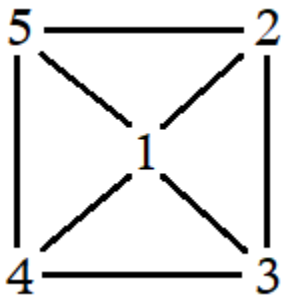
Visited = [F **T** **T** **T** **T** **T** F **T**]

The neighbors of 5 are 4 & 6. Among them, 5 is visited and hence we will visit 6 and the value of the 6th index in the visited array is updated to True.

Visited = [F **T** **T** **T** **T** **T** **T** **T**]

The neighbors of 6 are 5 & 7. Both are visited and thus we have visited all the nodes in depth-way. **This is called DFS.**

Q) Visited = [F F F F F F]
0 1 2 3 4 5



We will visit 1 and the value at 1st index in the visited array is updated to True. Its neighbors are 5, 4, 3 & 2.

Visited = [F **T** F F F F]

Now, we will visit 5 and the 5th element in visited array is updated to True. Its neighbors are 4, 1 & 2.

Visited = [F **T** F F F **T**]

Since 1 is already visited, we will visit 4 and 4th element in visited array is updated to True. Its neighbors are 5, 3 & 1.

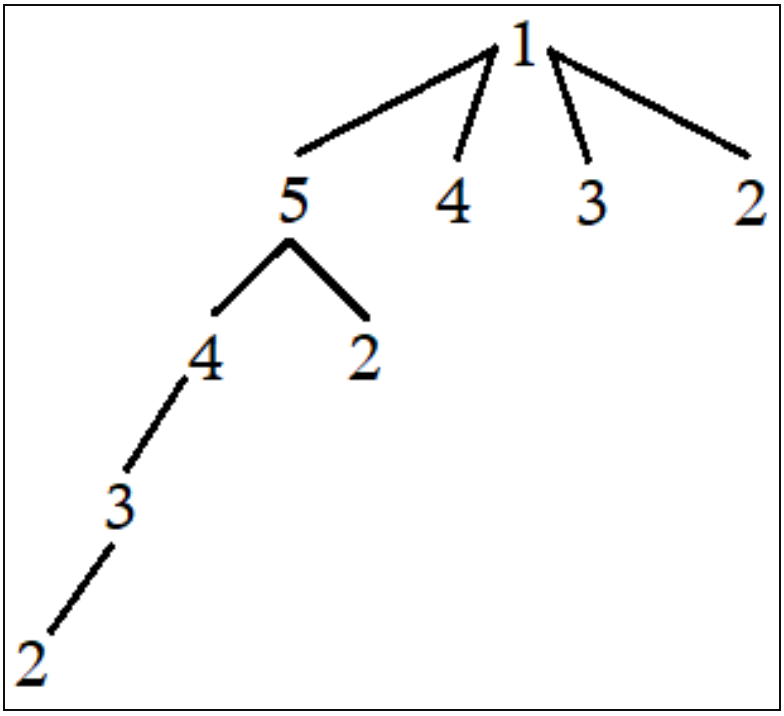
Visited = [F **T** F F **T** **T**]

Now, since 5 is visited, we will visit 3 and the 3rd element in visited array is updated to True. Its neighbors are 4, 2 & 1.

Visited = [F **T** F **T** **T** **T**]

Now, since 4 is visited, we will visit 2 and 2nd element in visited array is updated to True. its neighbors are 5, 3 & 1.

Visited = [F **T** **T** **T** **T** **T**]



CODE:

```
graph = dict()

def dfs(src, visited):
    visited[src] = True

    for neibour, wt in graph[src]:
        if not visited[neibour]:
            dfs(neibour, visited)

def addEdge(u, v, weight, directed):
    if u not in graph:
        graph[u] = list()

    graph[u].append((v, weight))

    if not directed:
        if v not in graph:
            graph[v] = list()
        graph[v].append((u, weight))

if __name__ == "__main__":

    addEdge(1, 2, 1, False)
    addEdge(0, 3, 10, False)
    addEdge(2, 3, 8, False)
    addEdge(3, 2, 11, False)
    addEdge(2, 5, 133, False)


    bfs(0)

    print("dfs traversal")
    visited = [False] * 10005
    dfs(0, visited)
```

Time Complexity for this code would be ***O(V+E)***. The space complexity would be ***O(V*V)***.

S.NO	BFS – Binary First Search	DFS – Depth First Search
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
3.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
4.	BFS considers all neighbours first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

LeetCode: 200 Number of Islands

Given an m x n 2d grid map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input: grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

Output: 1

Example 2:

Input: grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

Output: 3

In these types of questions, we can do either BFS or DFS.

```
class Solution:
    def dfs(self, x, y):
        self.visited[x][y] = True

        if x + 1 < self.n and not self.visited[x+1][y] and self.grid[x+1][y] == '1':
            self.dfs(x + 1, y)

        if x - 1 >= 0 and not self.visited[x-1][y] and self.grid[x-1][y] =='1':
            self.dfs(x - 1, y)

        if y + 1 < self.m and not self.visited[x][y+1] and self.grid[x][y+1]:
            self.dfs(x, y + 1)

        if y - 1 >= 0 and not self.visited[x][y-1] and self.grid[x][y-1]:
            self.dfs(x, y - 1)


    def numIslands(self, grid: List[List[str]]) -> int:
        self.grid = grid

        self.n = len(self.grid)
        self.m = len(self.grid[0])
```

```
        self.visited = [[False for _ in range(self.m)] for _ in range(self.n)]

        cnt = 0
        for i in range(self.n):
            for j in range(self.m):
                if self.grid[i][j] == "1" and not self.visited[i][j]:
                    self.dfs(i, j)
                    cnt += 1

        return cnt
```

Instead of writing 4 if conditions,

```
class Solution:
    def dfs(self, x, y):
        self.visited[x][y] = True

        dxy = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        for dx, dy in dxy:
            new_x = x + dx
            new_y = y + dy
            if new_x >= 0 and new_x < self.n and new_y >= 0 and new_y < self.m:
                if not self.visited[new_x][new_y] and self.grid[new_x][new_y] == '1'
:
                    self.dfs(new_x, new_y)

    def numIslands(self, grid: List[List[str]]) -> int:
        self.grid = grid

        self.n = len(self.grid)
        self.m = len(self.grid[0])

        self.visited = [[False for _ in range(self.m)] for _ in range(self.n)]

        cnt = 0
        for i in range(self.n):
            for j in range(self.m):
                if self.grid[i][j] == "1" and not self.visited[i][j]:
                    self.dfs(i, j)
                    cnt += 1
        return cnt
```

Resources:

- https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/
- Depth First Search or DFS for a Graph - GeeksforGeeks
- Breadth First Search or BFS for a Graph - GeeksforGeeks