# Breadth First Search (BFS): Level Order Traversal

**Problem Statement:** Given an undirected graph, return a vector of all nodes by traversing the graph using breadth-first search (BFS).

Approach : We shall use the queue datastructure to traverse the graph.

**Time Complexity :** O(V+E) because, in the worst-case scenario, we need to visit all vertices and edges of the graph once during the traversal.

In a graph with V vertices and E edges, the worst-case scenario is when every vertex is connected to every other vertex, creating a fully connected graph. In this case, the number of edges is E=V(V-1)/2 (since each vertex is connected to V-1 other vertices), so we can rewrite the time complexity as:

O(V + V(V-1)/2)

Simplifying this expression, we get:

O(V^2)

BFS starts by visiting the starting vertex (let's call it s), and then visits all vertices at a distance of one edge from s. Then, it visits all vertices at a distance of two edges from s, and so on, until it has visited all vertices in the graph.

# Depth First Search (DFS)

**Problem Statement:** Given an undirected graph, return a vector of all nodes by traversing the graph using depth-first search (DFS).

Approach : Involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

**Time Complexity :** The time complexity will be O(V + E), where E is the number of edges in the graph. This is because the DFS algorithm visits each vertex and each edge once in worst case.

# Number of Provinces

Problem Statement: Given an undirected graph with V vertices. We say two vertices u and v belong to a single province if there is a path from u to v or v to u. Your task is to find the number of provinces.

Approach :

We can use any of the traversals to solve this problem because a traversal algorithm visits all the nodes in a graph. In any traversal technique, we have one starting node and it traverses all the nodes in the graph. Suppose there is an ‘N’ number of provinces so we need to call the traversal algorithm ‘N‘ times, i.e., there will be ‘N’ starting nodes. So, we just need to figure out the number of starting nodes.

**Time Complexity :** O(V+E)

Note : If they give the Adjacency matrix as input graph , we can directly solve that or we may convert that adjacency matrix to adjacency list and proceed.

# Rotten Oranges

**Problem Statement:** Given a grid of dimension N x M where each cell in the grid can have values 0, 1, or 2 which has the following meaning:

0: Empty cell

1: Cells have fresh oranges

2: Cells have rotten oranges

We have to determine what is the minimum time required to rot all oranges. A rotten orange at index [i,j] can rot other fresh oranges at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right) in unit time.

**Approach :**

The idea is to first identify all the rotten oranges in the grid and add them to a queue. Then, we perform a BFS on the grid by dequeuing each rotten orange from the queue, and adding its adjacent fresh oranges to the queue while marking them as rotten(next layer of rotten oranges). While doing so, we keep a counter of the number of minutes that have elapsed.

We repeat this process until all the fresh oranges have been marked as rotten or until there are no more fresh oranges left.

If all the fresh oranges have been marked as rotten, we return the number of minutes that have elapsed. Otherwise, if there are still fresh oranges left, it means that they are not reachable from any rotten orange, and hence are impossible to become rotten. In this case, we return -1.

**Time Complexity :** O(N), where N is the total number of cells in the grid.

The reason for this is that in the worst case scenario, every cell in the grid needs to be visited to determine the time it takes for all the oranges to rot or if there are any fresh oranges left.

# Flood Fill Algorithm – Graphs

**Problem Statement:** An image is represented by a 2-D array of integers, each integer representing the pixel value of the image. Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, “flood fill” the image.

To perform a “flood fill”, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same colour as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same colour as the starting pixel), and so on. Replace the colour of all of the aforementioned pixels with the newColor.

**Approach :** Check for the neighbours of the respective pixel that has the same initial colour and has not been visited or coloured. DFS call goes first in the depth on either of the neighbours.

We can either use a separate visited matrix or ans matrix only (we can check if the current cell is already visited or not i.e filled with new color or not , the exceptional case would be when newcolor and old color are same )

**Time Complexity :** O(N), where N is the number of pixels in the input image. This is because each pixel is visited at most once, and the time required to process each pixel is constant.

In the worst case, every pixel in the image may need to be processed by the algorithm.

# Detect Cycle in an Undirected Graph (using BFS)

**Problem Statement:** Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.