# 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.

**Approach :**

The intuition is that we start from a node, and start doing BFS level-wise, if somewhere down the line, we visit a single node twice, it means we came via two paths to end up at the same node.

Push the pair of the source node and its parent data (<source, parent>) in the queue, and mark the node as visited.

The case where neighbour is not a paraent and visited already is the satisfactory condition for the cycle .

**Time Complexity :** O(V + E ) , normal BFS traversal

# Detect Cycle in an Undirected Graph (using DFS)

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

**Approach :** The intuition is that we start from a source and go in-depth, and reach any node that has been previously visited in the past; it means there’s a cycle.

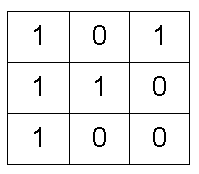
Same condition as above , the case where neighbour is not a paraent and visited already is the satisfactory condition for the cycle .

# Distance of Nearest Cell having 1

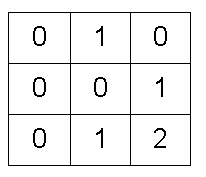
**Problem Statement**: Given a binary grid of N\*M. Find the distance of the nearest 1 in the grid for each cell.

The distance is calculated as |i1 – i2| + |j1 – j2|, where i1, j1 are the row number and column number of the current cell, and i2, j2 are the row number and column number of the nearest cell having value 1.

**Input:**



**Output:**



**Approach :**

The intuition is that BFS will take a step from cells containing 1 and will reach out to all zeros that are at a distance of one. Apparently, we can say that the nearest 1 to the 0s is at a distance of one. Again if we take another step, we will reach the next set of zeros, for these zeros 1 is at a distance of two. If we continue the same, till we can go, we can reach all the 0’s possible.

We will choose the BFS algorithm as it moves step by step and we want all of them to traverse in a single step together so that we can have a minimum count with us.

In the BFS queue we shall take (x, y , distance ) as triplet datastructure.

**Question :** Why are we not considering diagonal sides while exploring ?

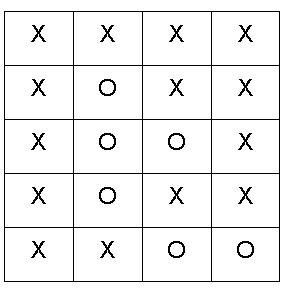
**Answer :** The idea is to cover only the steps which are at a distance of 1, which are the ones nothing but adjacent to that in 4 directions.

**Time Complexity:** For the worst case, the BFS function will be called for (N x M) nodes.

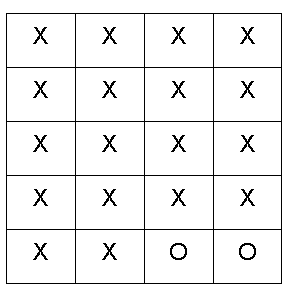
# Surrounded Regions | Replace O’s with X’s

**Problem Statement:** Given a matrix mat of size N x M where every element is either ‘O’ or ‘X’. Replace all ‘O’ with ‘X’ that is surrounded by ‘X’. An ‘O’ (or a set of ‘O’) is considered to be surrounded by ‘X’ if there are ‘X’ at locations just below, just above just left, and just right of it.

Input :



Output :



**Approach :**

The boundary elements in the matrix cannot be replaced with ‘X’ as they are not surrounded by ‘X’ from all 4 directions. This means if ‘O’ (or a set of ‘O’) is connected to a boundary ‘O’ then it can’t be replaced with ‘X’.

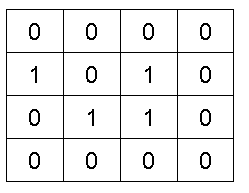
The intuition is that we start from boundary elements having ‘O’ and go through its neighboring Os in 4 directions and mark them as visited to avoid replacing them with ‘X’.

**Time Complexity :** For the worst case, every element will be marked as ‘O’ in the matrix, and the DFS function will be called for (N x M) nodes.

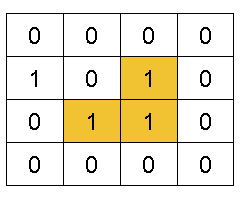
# Number of Enclaves

**Problem Statement:** You are given an N x M binary matrix grid, where 0 represents a sea cell and 1 represents a land cell. A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid. Find the number of land cells in the grid for which we cannot walk off the boundary of the grid in any number of moves.

**Input:**



**Output :** 3 (from the highlighted 3 cells , we can not walkout of the grid )



**Approach :**

The land cells present in the boundary cannot be counted in the answer as we will walk off the boundary of the grid. Also, land cells connected to the boundary land cell can never be the answer.

The intuition is that we need to figure out the boundary land cells, go through their connected land cells and mark them as visited. The sum of all the remaining land cells will be the answer.

**Time Complexity :** For the worst case, every element will be marked as visited in the matrix, and the DFS function will be called for (N x M) nodes.

# Word Ladder – I

Given are the two distinct words startWord and targetWord, and a list denoting wordList of unique words of equal lengths. Find the length of the shortest transformation sequence from startWord to targetWord.

In this problem statement, we need to keep the following conditions in mind:

* A word can only consist of lowercase characters.
* Only one letter can be changed in each transformation.
* Each transformed word must exist in the wordList including the targetWord.
* startWord may or may not be part of the wordList

Note: If there’s no possible way to transform the sequence from startWord to targetWord return 0.

Example 1:

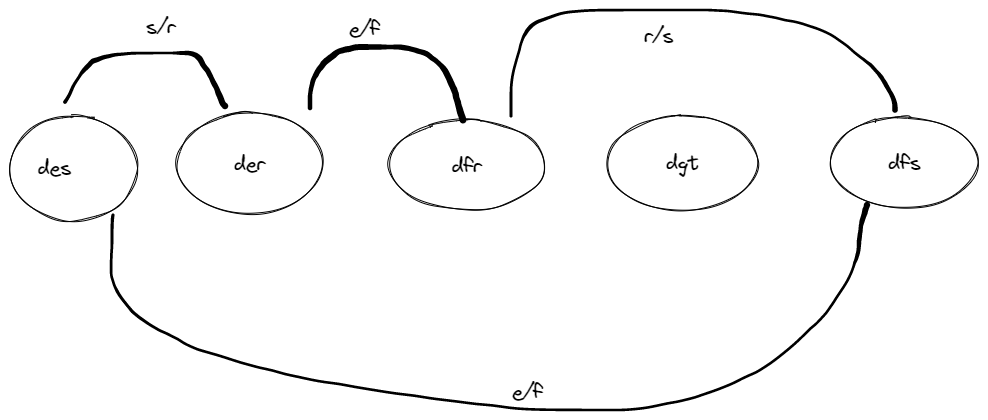
Input:

wordList = {"des","der","dfr","dgt","dfs"}

startWord = "der", targetWord = "dfs"

Output:

3



Explanation:

The length of the smallest transformation sequence from "der" to

"dfs" is 3 i.e. "der" -> (replace ‘e’ by ‘f’) -> "dfr" -> (replace ‘r’ by ‘s’) -> "dfs". So, it takes 3 different strings for us to reach the targetWord. Each of these strings are present in the wordList.

Example 2:

Input:

wordList = {"geek", "gefk"}

startWord = "gedk", targetWord= "geek"

Output:

2

Explanation:

The length of the smallest transformation sequence

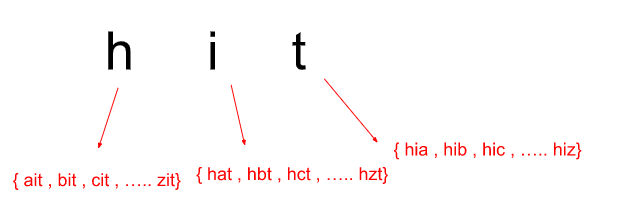
from "gedk" to "geek" is 2 i.e. "gedk" -> (replace ‘d’ by ‘e’) -> "geek"

So, it takes 2 different strings for us to reach the targetWord.

Each of these strings are present in the wordList.

**Approach :**

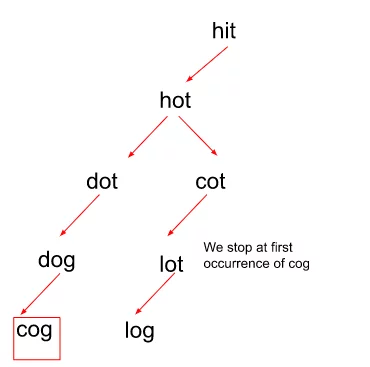
In Brute force, we just simply replace the startingWord character by character and then check whether the transformed word is present in the wordList. If a word is present in the wordList, we try replacing another character in that word by again following similar steps as above, in order to attain the targetWord. We do this for all the characters in the startWord and then eventually return the minimum length of transforming the startWord to targetWord.



Now, to make this algorithm a little less time-consuming and easier, we implement this using a BFS traversal technique.

Let us take an words = { “hot” , “dot” , “dog” , “lot” , “log” , “cog” }

startWord = “hit” , endWord = “cog”



At a given level , we are supposed to do only one transformation of character , if the transformed word exists in the given words and is not yet visited , we shall push it to the queue.

Question : Why are we erasing the words from the set ?

Answer : Here erasing means marking as visited , If a word is valid and unvisited we would erase so that , it does not come again for further transformations. (Ex : dot -> hot we are not supposed to go back .)

**Time Complexity:** O(N \* M \* 26)

Where N = size of wordList Array and M = word length of words present in the wordList.